Computational Geometry 43 (2010) 493–513

Contents lists available at ScienceDirect

Computational Geometry: Theory and Applications

www.elsevier.com/locate/comgeo

Star-quadtrees and guard-quadtrees: I/O-efficient indexes for fat triangulations and low-density planar subdivisions ☆

Mark de Berg [a,*], Herman Haverkort [a], Shripad Thite [b], Laura Toma [c]

[a] Department of Mathematics and Computer Science, Technische Universiteit Eindhoven, The Netherlands
[b] California Institute of Technology, Center for the Mathematics of Information, USA
[c] Department of Computer Science, Bowdoin College, Brunswick, ME, USA

ARTICLE INFO

Article history:
Received 14 October 2008
Received in revised form 31 August 2009
Accepted 13 November 2009
Available online 17 November 2009
Communicated by P. Widmayer

Keywords:
Quadtrees
I/O-efficient spatial indexes
Fat triangulations
Low-density subdivisions

ABSTRACT

We present a new I/O-efficient index structure for storing planar subdivisions. This so-called *guard-quadtree* is a compressed linear quadtree, which is provably efficient for map overlay, point location, and range queries in low-density subdivisions. In particular, we can preprocess a low-density subdivision with n edges, in $O(sort(n))$ I/Os, to build a guard-quadtree that allows us to:

(i) compute the intersections between the edges of two such preprocessed subdivisions in $O(scan(n))$ I/Os, where n is the total number of edges in the two subdivisions;
(ii) answer a single point location query in $O(\log_B n)$ I/Os, and k batched point location queries in $O(scan(n) + sort(k))$ I/Os; and
(iii) answer range queries for any constant-complexity query range Q in $O(\frac{1}{\varepsilon}(\log_B n) + scan(k_\varepsilon))$ I/Os for every $\varepsilon > 0$, where k_ε is the number of edges of the subdivision within distance $\varepsilon \cdot \mathrm{diam}(Q)$ from Q.

For the special case where the subdivision is a fat triangulation, we show how to obtain the same results with an ordinary (uncompressed) linear quadtree, which we call the *star-quadtree*. The star-quadtree is fully dynamic and needs only $O(\log_B n)$ I/Os per update.
Our algorithms and data structures improve on the previous best known bounds for overlaying general subdivisions, both in the number of I/Os and space usage. The constants in the asymptotic bounds are small, which makes our results applicable in practice. Moreover, our algorithms are simpler than previous approaches and almost all of them are cache-oblivious.

1. Introduction

Motivation. Traditionally, algorithm design and analysis assumes each elementary operation takes approximately the same amount of time. As a result, the number of such operations is a measure of the total running time of the algorithm. Unfortunately, this simplifying assumption is invalid when the algorithm operates on data stored on disk: reading data from or writing data to disk can be a factor 1,000,000 or more slower than doing an operation on data that is already present in

☆ M.d.B. and S.T. were supported by the Netherlands' Organisation for Scientific Research (NWO) under project No. 639.023.301. L.T. was supported by NSF award No. 0728780.

* Corresponding author.
E-mail addresses: mdberg@win.tue.nl (M. de Berg), cs.herman@haverkort.net (H. Haverkort), shripad@caltech.edu (S. Thite), ltoma@bowdoin.edu (L. Toma).

0925-7721/$ – see front matter
doi:10.1016/j.comgeo.2009.11.001

main memory. Thus, when the data is stored on disk, it is usually much more important to minimize the number of disk accesses, rather than to minimize the CPU computation time.

This importance of input–output or I/O operations has led to the study of so-called I/O-efficient algorithms, which are also known as external-memory or out-of-core algorithms. The I/O-model introduced by Aggarwal and Vitter [1] has become a popular tool for analyzing I/O-efficient algorithms. In this two-level memory model, a computer has an internal memory of size M and an arbitrarily large disk. The data on disk is stored in blocks of size B. Whenever an algorithm wants to work on data not present in internal memory, the block(s) containing the data are read from disk. The *I/O-complexity* of an algorithm is the number of I/Os it performs, that is, the number of block transfers between internal memory and the disk. In this model, scanning—reading a set of n consecutive items from disk—can be done in $scan(n) = \Theta(n/B)$ I/Os, and sorting takes $sort(n) = \Theta((n/B)\log_{M/B}(n/B))$ I/Os [1]. Algorithms that are *cache-aware* may use knowledge of M and B and take control over which blocks are evicted from memory to make place for others. Algorithms that are *cache-oblivious*, however, cannot use knowledge of M and B and have no control over which blocks are evicted from memory to make place for others. Nevertheless it is possible to analyse the number of I/Os needed by a cache-oblivious algorithm as a function of n, M and B, assuming that the operating system decides which blocks are kept in memory using a paging strategy (caching policy) that satisfies mild assumptions [16]. A cache-oblivious algorithm with good bounds on the number of I/Os automatically exhibits good I/O-behaviour over all levels of a multilevel memory hierarchy. This is because the algorithm is not tuned to the values of M and B of any particular memory level, and the bounds on the number of I/Os hold for the different values of M and B of all memory levels simultaneously.

One of the main application domains for I/O-efficient algorithms is that of geographic information systems (GIS), because GIS typically work with massive amounts of data, and loading all of it into memory is often infeasible. In GIS the data for a particular geographic region is stored as a number of separate thematic layers. There can be a layer storing the road network, a layer storing the river network, a layer storing a (planar) subdivision of the region according to land usage or soil type, and so on. These layers are then stored in a suitable data structure—or *index* in GIS terminology—such that certain operations can be performed efficiently. Three important operations are map overlay, point location, and range searching, which are described next.

Map overlay is used to combine the information from two layers in a GIS, for example to find the crossings between the river network and the road network. The problem of map overlay can be formulated as a red–blue intersection problem: given a set of non-intersecting blue segments and a set of non-intersecting red segments in the plane—these sets contain the segments forming the two layers, respectively—compute all intersections between the red and blue segments. If the layers represent subdivisions, then the map-overlay problem is to construct the subdivision resulting from overlaying the two layers. This involves computing the intersections between the edges of the two subdivisions—these will be vertices in the overlay. Computing the complete overlay also involves computing the faces of the new subdivision. For triangulations this is fairly easy; for arbitrary subdivisions it can be more difficult, especially if the subdivisions can have holes. Moreover, what is exactly needed depends on the required representation of the overlay. In this paper we restrict our attention to computing the edge-intersections; with a slight abuse of terminology we will still refer to this as the map-overlay problem.

Point location means to locate a point in a subdivision stored in a single layer. A point location query asks to report, for a query point q, the face of a given subdivision containing q.

Finally, a *range searching* query asks to report everything from the layer that intersects (that is, lies fully or partially within) a query range Q.

Even though spatial databases and GIS are main application areas for I/O-efficient algorithms, no indexes are known that are provably I/O-efficient for all these operations. This is the goal of our paper: to develop index data structures for planar subdivisions that support the above operations in a provably I/O-efficient manner.

Background. Most research into I/O-efficient algorithms has focused on algorithms that are efficient in the worst case. However, worst-case inputs are often artificial constructions that do not occur in practical applications. Algorithms that are designed to be efficient in the worst case might not perform optimally on realistic inputs. In computational geometry, the desire to design algorithms that are provably efficient in practice has led to the study of input models where inputs are assumed to have properties that make them better resemble realistic inputs [13]. The most common assumption in such models is that the objects are *fat*, that is, they are not arbitrarily long and narrow. Fatness has been studied extensively in computational geometry in recent years, and solutions to many fundamental problems have been improved by exploiting fatness and related notions; see [11,13] and references therein.

In this paper, we consider two types of planar subdivisions: fat triangulations and low-density subdivisions. A *δ-fat triangulation* is a triangulation in which every angle is bounded from below by a fixed positive constant δ. A *λ-low-density subdivision* is a subdivision such that any disk D is intersected by at most λ edges whose length is at least the diameter of D, for some fixed constant λ. We believe these two types of subdivisions represent most subdivisions encountered in practice, for reasonable values of δ and λ. For a summary of realistic input models we refer the reader to [13]. We obtain a simple indexing structure, that is provably I/O-efficient, for map-overlay operations, point location, and range queries for such subdivisions. It is known that any set of disjoint δ-fat triangles in the plane has density $O(1/\delta)$ [13]. Thus the results for low-density subdivisions can be used for fat triangulations. Nevertheless we also consider the special class of fat triangulations separately, because we were able to obtain simpler algorithms and to support efficient updates in this case.

One of the best known and widely used indexing structures is the quadtree [21]. A particularly useful variant for external-memory applications is the so-called *linear quadtree*, which was introduced by Gargantini [17]. The linear quadtree is a quadtree variant where only the leaf regions are stored, and not the internal nodes. To facilitate a search in the quadtree, a linear order is defined on the leaves based on some space-filling curve; then a B-tree is constructed on the leaves using this ordering—see Section 2 for details. The idea of using linear quadtrees to store planar subdivisions has been used by Hjaltason and Samet [20]. They present algorithms for constructing (or "bulk-loading", as it is often called in GIS) the quadtree, for insertions, and for bulk-insertions. Although their experiments indicate their method performs well in practice, it has several disadvantages. First, the I/O-complexity of their algorithms is analysed in terms of various parameters that depend on the data and the algorithm in a way that is not straightforward. In particular, the performance of their algorithms does not seem to be worst-case optimal. Second, the stopping rule for splitting quadtree cells is based on two user-defined parameters (the maximum depth and a so-called splitting threshold), and so the method is not fully automatic.

Our results. In this paper we show how to overcome these disadvantages for fat triangulations and low-density subdivisions. We present improved and simplified algorithms for map overlay, point location, and range searching in external memory for these two types of inputs. Our results are based on a quadtree which we define to ensure that each leaf intersects only a constant number of edges of the subdivision. The quadtree has only $O(n)$ leaves, which can be constructed efficiently.

For fat triangulations, our *star-quadtree* is defined by recursively splitting the unit square into quadrants until all edges that intersect a cell are incident to a common vertex. We prove that this stopping rule yields a quadtree of linear size. Nevertheless, due to the potentially large depth of the quadtree, it is still difficult or impossible to build the quadtree I/O-efficiently by distributing the edges from the root down into the quadtree while splitting nodes as needed. Fortunately our stopping rule makes a completely different approach possible: we give an algorithm that is simple and elegant—simpler than the algorithm of Hjaltason and Samet [20]—and uses only $O(sort(n))$ I/Os to build the tree.

For low-density subdivisions we continue splitting until each cell contains at most a constant number of bounding-box vertices of any edge. This stopping rule leads to cells with a constant number of edges, but the number of cells cannot be bounded. Therefore we combine the ideas of compressed quadtrees and linear quadtrees to obtain a *linear compressed quadtree*, rather than a regular quadtree. We show that with the stopping rule just defined, the compressed quadtree, which we call the *guard-quadtree*, has linear size. We also give a construction algorithm that uses only $O(sort(n))$ I/Os. Our guard quadtree can actually be used for any low-density set of segments—the segments need not form a proper subdivision and may even intersect.

Once we have proved that these quadtrees have linear size and each leaf region intersects a constant number of edges, most of our other results follow almost immediately. Overlaying two subdivisions—more precisely, red–blue intersection—is achieved by a simple merge of the ordered lists of quadtree leaves, taking $O(scan(n))$ I/Os. Point location can be done in $O(\log_B n)$ I/Os by searching in the B-tree built on top of the list of quadtree leaves, and k batched point location queries can be done in $O(scan(n) + sort(k))$ I/Os by sorting the points along the space-filling curve and merging the sorted list with the list of quadtree leaves. (The point location result assumes, of course, that the input is a proper subdivision.) For any $\varepsilon > 0$, all triangles intersecting a given query range Q can be reported in $O(\frac{1}{\varepsilon}(\log_B n) + scan(k_\varepsilon))$ I/Os, where k_ε is the number of edges within distance $\varepsilon \cdot \mathrm{diam}(Q)$ of Q. To obtain the same result on range queries for low-density subdivisions, we apply a pruning technique that was defined by de Berg and Streppel [14]. In [14], this technique was defined in a top-down fashion; to obtain an I/O-efficient solution we show how to implement this technique bottom-up. The results for map overlay apply to pairs of fat triangulations, low-density subdivisions, or low-density sets of line segments, as well as to mixed pairs of maps of these types, and to the output of range queries on these maps: one can obtain the overlay of two maps restricted to a certain query range in $O(\frac{1}{\varepsilon}(\log_B n) + scan(k_\varepsilon))$ I/Os. The structure for fat triangulations can be made fully dynamic, except that updates will then take amortized $O(\log_B n + \frac{1}{B}\log^2 n)$ I/Os.

Comparison with previous results. Arge et al. [4] showed how to solve the map-overlay problem in $O(sort(n) + scan(k))$ I/Os, where k is the number of intersections. Even though this is optimal in the worst case, it is not satisfactory for several reasons. First, as pointed out by the authors, their solution is complicated, and so its practical value is unclear. Second, the algorithm uses $\Theta(n \log_{M/B}(n/B))$ storage, that is, $\Theta((n/B)\log_{M/B}(n/B))$ disk blocks. A randomized solution for computing the intersections in a set of line segments—which could also be used for map overlay—is described by Crauser et al. [10]. They give the same expected I/O-bound of $O(sort(n) + scan(k))$ I/Os and linear space under some (realistic) assumptions for M, B and n. Recently, Arge et al. [2] gave an optimal (but complicated) cache-oblivious algorithm that uses $O(sort(n) + scan(k))$ I/Os and linear space. Although the I/O-complexity of the above algorithms is optimal for general sets of line segments, there are important special cases for which this is not clear. For example, the overlay of two suitably stored connected subdivisions can be computed in $O(n+k)$ time in internal memory [15]. Our work implies that for fat triangulations or low-density subdivisions a similar result is possible in external memory, since we can compute their overlay in $O(scan(n))$ I/Os if they are stored in our new indexing structure. Note that if we also include the time to construct the indexes, we get the same asymptotic bounds on the number of I/Os as Arge et al. [4] and Crauser et al. [10], while being deterministic and using only linear space, and using much simpler algorithms.

An optimal static structure for point location in general planar subdivisions was already given by Goodrich et al. [18] for the standard I/O-model and by Bender et al. [5] for the cache-oblivious model. Batched point location can be done using $O(sort(n+k))$ I/Os in the I/O-model using the algorithm by Arge et al. [4]; with $O(scan(n) + sort(k))$ I/Os, our bound on

preprocessed maps is slightly better. Our result on dynamization is new as far as we know: the best known dynamic I/O-efficient point location structures use $O(\log_B^2 n)$ I/Os per query [3] in the I/O-model, while we support queries and updates in fat triangulations in $O(\log_B n)$ I/Os. For range queries the oBAR-B-tree by Streppel and Yi [22] gives a better bound, $O(\frac{1}{\varepsilon} + \log_B n + scan(k_\varepsilon))$, but they do not support subsequent overlay operations in $O(scan(k_\varepsilon))$ I/Os.

Unlike some of the data structures mentioned above, all our data structures and query algorithms are cache-oblivious. Our construction and update algorithms for triangulations can be made cache-oblivious, except that updates will then take amortized $O(\log_B n + \frac{1}{B}\log^2 n)$ I/Os. These results constitute the first results for cache-oblivious batched point location, dynamic point location and range searching.

Our contribution can thus be summarized as follows. By exploiting the properties of fat triangulations and low-density subdivisions we are able to obtain very simple general-purpose indexing structures that yield improved I/O-bounds (compared to the best known bounds obtained without the fatness and low-density assumptions) for a variety of problems.

2. Fat triangulations: the star-quadtree

In this section we describe our star-quadtree for fat triangulations. A δ-fat triangulation is a triangulation consisting of δ-fat triangles, that is, triangles all of whose angles are at least δ for some fixed constant $\delta > 0$. We assume that our input is a δ-fat triangulation $\mathcal{F}$ of the unit square.

A *constant-complexity query range* Q is a subset of the plane bounded by a closed curve ∂Q with the following properties:

1. ∂Q has constant description size;
2. ∂Q has only a constant number of local extrema in x- and y-direction;
3. for any axis-parallel square σ it can be decided in constant time whether σ intersects Q.

In this section we show the following:

Theorem 2.1. *Let $\mathcal{F}$ be a δ-fat triangulation with n edges. Assume that the memory size M is at least c/δ^3, for a sufficiently large constant $c > 0$. We can construct, in $O(sort(n/\delta^2))$ I/Os, a linear quadtree for $\mathcal{F}$ that stores $O(n/\delta^2)$ cell-edge intersections. We call this linear quadtree the* star-quadtree *for $\mathcal{F}$. With the star-quadtree we can perform the following operations*:

(i) **Map overlay:** *Given two δ-fat triangulations with n triangles in total, each stored in a star-quadtree, we can find all pairs of intersecting triangles in $O(scan(n/\delta^2))$ I/Os.*
(ii) **(Batched) point location:** *For any query point p we can find the triangle of $\mathcal{F}$ that contains p in $O(scan(1/\delta) + \log_B(n/\delta))$ I/Os, and for any set P of k query points we can find the triangle of $\mathcal{F}$ that contains each point $p \in P$ in $O(scan(n/\delta^2) + sort(k))$ I/Os.*
(iii) **Range searching:** *For any constant-complexity query range Q and any constant $\varepsilon > 0$ we can find the triangles that intersect Q in $O(\frac{1}{\varepsilon}scan(1/\delta) + \frac{1}{\varepsilon}\log_B(n/\delta) + scan(k_\varepsilon/\delta^2))$ I/Os, where k_ε is the number of triangles that lie at distance at most $\varepsilon \cdot \mathrm{diam}(Q)$ from Q.*
(iv) **Updates:** *Inserting a vertex, moving a vertex, deleting a vertex, and flipping an edge can all be done in $O(\frac{1}{\delta^4}\log_B(n/\delta))$ I/Os.*

The same bounds hold in the cache-oblivious model, except that updates then take $O(\frac{1}{\delta^4}(\log_B(n/\delta) + \frac{1}{B}\log^2(n/\delta)))$ I/Os amortized.

The requirement that M is at least c/δ^3 applies only to the construction and updates of the data structure. Once the data structure has been constructed, map overlay can be done within the bounds indicated if M is at least c/δ, for a large enough constant c. As will be clear, the requirements on the total cache size M arise from the need to store all the $O(1/\delta)$ triangles incident on a single vertex and a so-called "local" quadtree of size $O(1/\delta^3)$ for this vertex neighborhood storing $O(1/\delta^3)$ cell-edge intersections, in internal memory. If the maximum vertex degree is smaller, or if a better upper bound on the number of cell-edge intersections can be obtained, then the minimum cache size required to achieve the claimed I/O-efficiency will be correspondingly lower.

2.1. The quadtree subdivision for fat triangulations

A quadtree is a hierarchical subdivision of the unit square into quadrants, where the subdivision is defined by a criterion to decide what quadrants are subdivided further, and what quadrants are leaves of the hierarchy. A *canonical square* is any square that can be obtained by recursively splitting the unit square into quadrants. Thus, the sides of a canonical square have length of the form 2^{-s} for some integer $s \geqslant 0$. For a canonical square σ, let $\mathrm{parent}(\sigma)$ denote its parent, that is, the canonical square that contains σ and has twice its side length. The leaves of the quadtree form the quadtree subdivision; that is, a *quadtree* for a set of objects in the unit square is a subdivision into disjoint canonical squares, which we call *quadtree cells*, such that each cell obeys the stopping rule while its parent does not. The stopping rule we use is as follows:

Stopping rule for fat triangulations: Stop splitting a cell σ when all edges intersecting σ are incident on a common vertex.

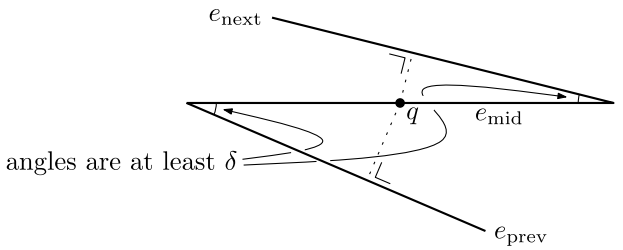

Fig. 1. The distance from q to e_{prev} and e_{next} is at least $\sin(\delta)|e_{\text{mid}}|/2$. Points on e_{mid} to the right of q have a larger distance to e_{prev}, points on e_{mid} to the left of q have larger distance to e_{next}.

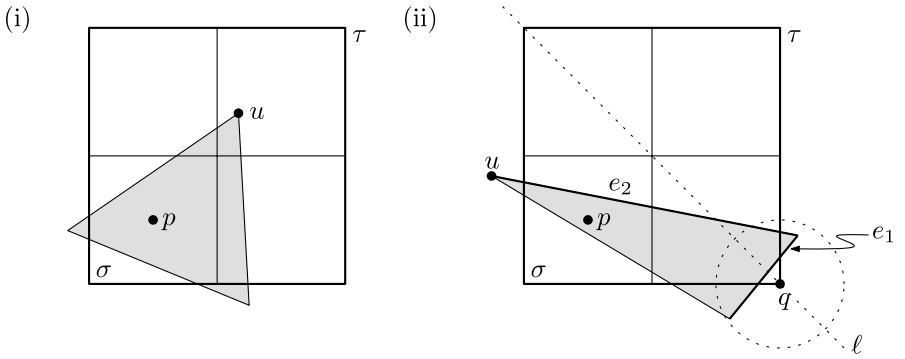

Fig. 2. Two cases in the proof of Lemma 2.3.

Note that the stopping rule includes the case were σ is not intersected by any edges. To prove that the stopping rule gives a linear number of cells, we need the following lemmas.

Lemma 2.2. *Let e_{prev}, e_{mid}, and e_{next} be three edges that form a path in a δ-fat triangulation (e_{prev} and e_{next} may share an endpoint, in which case the path is a triangle). Let $|e_{\text{mid}}|$ be the length of e_{mid}, and consider a square τ that intersects all three edges. Then* $\text{diam}(\tau) \geqslant \sin(\delta)|e_{\text{mid}}|/2$.

Proof. Consider the midpoint q of e_{mid}. The distance of q from e_{prev} and e_{next} is at least $d := \sin(\delta)|e_{\text{mid}}|/2$; see Fig. 1.

If $q \in \tau$, the lemma trivially follows because τ contains a segment s connecting q to e_{prev}, and the length of s is at least d.

If $q \notin \tau$, then τ must contain another point p on e_{mid}, which lies either to the left or to the right of q. In one case the distance from p to e_{prev} is at least d, while in the other case the distance from p to e_{next} is at least d. Hence, the lemma also holds if $q \notin \tau$. □

Lemma 2.3. *If σ is a leaf cell that does not contain a vertex of the triangulation $\mathcal{F}$, then its parent $\tau = \text{parent}(\sigma)$ intersects an edge e such that $\frac{1}{8}\sqrt{2}\,\text{diam}(\tau) < |e| \leqslant 2\,\text{diam}(\tau)/\sin(\delta)$.*

Proof. Since τ is not a leaf, there must be three edges that intersect it and are not all incident to the same vertex; more precisely there must be three edges e_{prev}, e_{mid} and e_{next} that intersect τ and form a path in the triangulation. By Lemma 2.2, we have $\sin(\delta)|e_{\text{mid}}|/2 \leqslant \text{diam}(\tau)$.

Next we show that τ intersects an edge e_{long} with $|e_{\text{long}}| > \frac{1}{4}\,\text{width}(\tau) = \frac{1}{8}\sqrt{2}\,\text{diam}(\tau)$. Consider the triangle $\triangle$ that contains the centre point p of σ. We distinguish two cases: either τ contains a vertex of $\triangle$, or not.

Case (i): τ contains a vertex u of $\triangle$—see Fig. 2(i).

Since σ does not contain u, we know that u is at distance at least $\frac{1}{4}\,\text{width}(\tau)$ from p. It follows that at least one of the edges of $\triangle$ incident to u (and thus intersecting τ) has length more than $|pu| \geqslant \frac{1}{4}\,\text{width}(\tau)$; let e_{long} be this edge.

Case (ii): τ does not contain a vertex of $\triangle$—see Fig. 2(ii).

Observe that τ must still intersect an edge e_1 of $\triangle$, since τ does not lie in the interior of a single triangle. If e_1 has length at least $\frac{1}{4}\,\text{width}(\tau)$, then let e_{long} be e_1. Otherwise e_1 must 'cut a corner' of τ and have both endpoints in a circle of radius $\frac{1}{4}\,\text{width}(\tau)$ around that corner q, as in Fig. 2(ii). Let ℓ be the line that contains the diagonal of τ incident to q, and let u be the vertex of $\triangle$ not incident to e_1. Since the two endpoints of e_1 lie on opposite

sides of ℓ, one of the two edges incident to u will intersect ℓ and, hence, τ. Since u lies outside τ by assumption and $\triangle$ contains p, that edge e_2 has length at least width(τ). Note that this also holds when, for example, q is a corner of σ. Let e_{long} be e_2.

We conclude that τ intersects an edge e_{mid} with length at most $2\,\text{diam}(\tau)/\sin(\delta)$ and an edge e_{long} (possibly equal to e_{mid}) with length more than $\frac{1}{8}\sqrt{2}\,\text{diam}(\tau)$. If we have $|e_{\text{long}}| \leqslant 2\,\text{diam}(\tau)/\sin(\delta)$ we select $e := e_{\text{long}}$. Otherwise, that is, if $|e_{\text{long}}| > 2\,\text{diam}(\tau)/\sin(\delta)$, we pick a point p on e_{long} in τ and a point q on e_{mid} in τ and traverse the line segment from p to q. Consider the edges crossed during this traversal. Consecutive edges belong to the same triangle and therefore differ in length by a factor between $\sin(\delta)$ and $1/\sin(\delta)$; hence, during this traversal we will eventually cross an edge e such that $\frac{1}{8}\sqrt{2}\,\text{diam}(\tau) < 2\,\text{diam}(\tau) < |e| \leqslant 2\,\text{diam}(\tau)/\sin(\delta)$. □

Now we are ready to prove bounds on the complexity of the quadtree subdivision.

Lemma 2.4. *Let $\mathcal{F}$ be a δ-fat triangulation of the unit square with n edges. Then the stopping rule above gives a quadtree subdivision with $O(n/\delta)$ cells, each intersecting at most $2\pi/\delta$ triangles.*

Proof. Since the degree of each vertex in $\mathcal{F}$ is at most $2\pi/\delta$, the stopping rule implies that each cell is intersected by at most that many triangles. It remains the prove the bound on the number of cells. To do so, we will charge cells either to vertices or to edges of the triangulation.

Charging to vertices. If a cell σ contains a vertex, we charge σ to this vertex. Each vertex lies in only one cell, so we have $O(n)$ cells charged to vertices in total.

Charging to edges. Any cell σ that does not contain a vertex is charged to an edge e that intersects the parent τ of σ and such that $\frac{1}{4}\sqrt{2}\,\text{diam}(\sigma) = \frac{1}{8}\sqrt{2}\,\text{diam}(\tau) < |e| \leqslant 2\,\text{diam}(\tau)/\sin(\delta) = 4\,\text{diam}(\sigma)/\sin(\delta)$, or, equivalently: $\frac{1}{4}\sin(\delta)|e| \leqslant \text{diam}(\sigma) < 2\sqrt{2}|e|$. Such an edge is guaranteed to exist by Lemma 2.3, since $\text{diam}(\sigma) = \text{diam}(\tau)/2$. To analyse the number of cells charged to any edge, we divide the cells charged to an edge e into size classes $E_i(e)$, for $i \in \{\lfloor\log(\frac{1}{4}\sin(\delta))\rfloor, \ldots, -2, -1, 0, 1\}$, where $E_i(e)$ contains cells σ charged to e with $2^i|e| \leqslant \text{diam}(\sigma) < 2^{i+1}|e|$. Consider one of these size classes $E_i(e)$. For $\sigma \in E_i(e)$ we have

$$\text{area}(\sigma) = \frac{1}{2}\,\text{diam}(\sigma)^2 \geqslant 2^{2i-1}|e|^2.$$

Let R_i be the Minkowski sum of e and a square with diameter $4 \cdot 2^{i+1}|e|$. We have

$$\text{area}(R_i) < \bigl(|e| + 4 \cdot 2^{i+1}|e|\bigr)\bigl(4 \cdot 2^{i+1}|e|\bigr) = \bigl(2^{i+3} + 2^{2i+6}\bigr)|e|^2.$$

Since the parent of each cell $\sigma \in E_i(e)$ intersects e, each such cell σ lies completely inside R_i. The cells $\sigma \in E_i$ are disjoint, so we can conclude that

$$|E_i| < \bigl(2^{i+3} + 2^{2i+6}\bigr)/2^{2i-1} = 2^4/2^i + 2^7.$$

By summing over all size classes we conclude that the number of cells charged to e is:

$$\sum_{i=\lfloor\log(\frac{1}{4}\sin(\delta))\rfloor}^{1} \bigl|E_i(e)\bigr| < \sum_{i=\lfloor\log(\frac{1}{4}\sin(\delta))\rfloor}^{1} \bigl(2^4/2^i + 2^7\bigr) = O(1/\delta).$$

By summing over all edges, we find that $O(n/\delta)$ cells are charged to edges. □

2.2. Storing the quadtree subdivision and the triangulation

We will store the quadtree subdivision defined above as a so-called *linear quadtree* [17]. To this end, we define an ordering on the leaf cells of the quadtree subdivision. The ordering is based on a space-filling curve defined recursively by the order in which it visits the quadrants of a canonical square. We will use the *Z-order space-filling curve* for this, which visits the quadrants in the order top left, top right, bottom left, bottom right, and within each quadrant, the Z-order curve visits its subquadrants recursively in the same order. Since the intersection of every canonical square with this curve is a contiguous section of the curve, this yields a well-defined ordering of the leaf cells of the quadtree subdivision. We call the resulting order the Z-order. Fig. 3 illustrates the ordering.

The Z-order curve not only orders the leaf cells of the quadtree subdivision, but also provides an ordering for any two points in the unit square—namely the Z-order of any two disjoint canonical squares containing the points. Assume a coordinate system in which the top left corner of the unit square is has coordinates $(0, 0)$, and the bottom right corner has coordinates $(1, 1)$. We assume that canonical squares are closed at the top and the left side, and open at the bottom and the right side. The Z-order of two points can now be determined as follows. For a point $p = (p_x, p_y)$ in $[0, 1)^2$, define its Z-index $Z(p)$ to be the value in the range $[0, 1)$ obtained by interleaving the bits of the fractional parts of p_y and p_x,

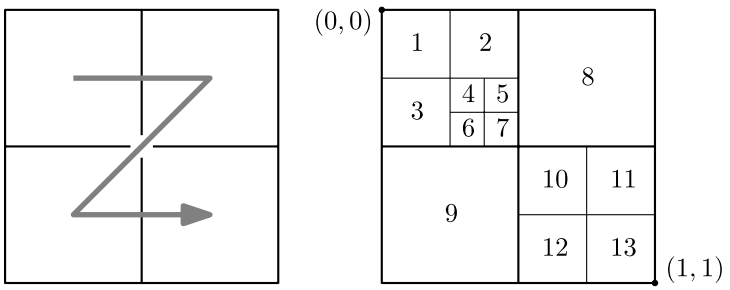

Fig. 3. Left: the order in which the Z-order curve visits the four quadrants of a canonical square. Right: an example of the resulting ordering.

starting with the first bit of p_y. The value $Z(p)$ is sometimes called the *Morton block index* of p. The Z-order of two points is now the same as the order of their Z-indices [20]. The Z-indices of all points in a canonical square σ form a subinterval $[z, z')$ of $[0, 1)$, where z is the Z-index of the top left corner of σ. Note that any subdivision of the unit square $[0, 1)^2$ into k leaf cells of a quadtree corresponds directly to a subdivision of the unit interval $[0, 1)$ of Z-indices into k subintervals.

A simple (but novel) way of storing a triangulation in a linear quadtree is now obtained by storing all the cell-triangle pairs $(\sigma, \triangle)$ such that σ intersects $\triangle$ in a B-tree [9], as follows: each cell-triangle pair $(\sigma, \triangle)$ of a cell σ corresponding to the Z-index interval $[z, z')$ is represented by storing triangle $\triangle$ with key z. With this way of storing the linear quadtree, the leaf cells of the quadtree are stored implicitly: each pair of consecutive different keys z_1 and z_2 represents the Z-index interval of a quadtree leaf cell. To obtain a cache-oblivious solution, we can use a cache-oblivious B-tree [5,6,8] to store the sequence of triangles.

In the remainder of this paper we will sometimes need to compute or compare Z-indices. Whether this takes constant time depends on the operations allowed by the model of computation. In any case, since we care mainly about I/O-efficiency, such computations do not affect the analysis of our algorithms.

2.3. Building the quadtree I/O-efficiently

A natural algorithm would be to build the quadtree top-down. We could take all the triangles and the unit square as the root, check if the stopping rule is satisfied by the root and, if not, distribute the triangles among the four children and recurse. Unfortunately, this algorithm takes $\Omega(n^2)$ time and $\Omega(n^2/B)$ I/Os in the worst case because the quadtree may have height $\Omega(n)$. One of the advantages of our stopping rule is that it allows us to compute the leaf cells directly, that is, using local computations instead of a top-down approach. We explain how to do this next.

For any vertex v of the given triangulation $\mathcal{F}$, let $star(v)$ be the *star* of v in $\mathcal{F}$; that is, the set of triangles of $\mathcal{F}$ that have v as a vertex. Recall that a canonical square is any square that can be obtained by recursively subdividing the unit square into quadrants.

For a set S of triangles inside the unit square, we say that a canonical square σ is *final in* S if it lies completely inside the union of the triangles in S and all edges from S that intersect σ are incident to a common vertex, while $\text{parent}(\sigma)$ intersects multiple edges of S that are not all incident to a common vertex. Thus, the cells of the quadtree subdivision we wish to compute for $\mathcal{F}$ are exactly the final canonical squares in $\mathcal{F}$.

Lemma 2.5. *Let $\triangle = (u, v, w)$ be a triangle of $\mathcal{F}$ and let σ be a canonical square intersecting $\triangle$. Then σ is final in $\mathcal{F}$ if and only if σ is final in $star(u)$, $star(v)$ or $star(w)$.*

Proof. Consider the canonical cell $\tau = \text{parent}(\sigma)$. Let γ_u denote the boundary of $star(u)$. The curve γ_u is a simple closed polygonal curve.

If σ is final in $star(u)$ then there exist three edges of $star(u)$ that intersect τ and are not all incident to the same vertex. Since σ is completely contained inside $star(u)$, it is separated from any edges of $\mathcal{F}$ outside $star(u)$ by γ_u. Hence σ is final in $\mathcal{F}$. The same argument implies that if σ is final in $star(v)$ or in $star(w)$, then it is final in $\mathcal{F}$.

As for the other direction, suppose σ is final in $\mathcal{F}$ and, without loss of generality, assume that all edges that intersect σ are incident to u. We distinguish three cases: τ intersects at least two, one, or zero edges incident to u.

Case (i): τ intersects at least two edges incident to u—see Fig. 4(i).
Since τ has been partitioned, it must intersect a third edge that is not incident to u; in particular it intersects an edge of γ_u. The square τ thus intersects three edges of $star(u)$ that are not all incident to a common vertex, and its child σ is final in $star(u)$.

Case (ii): τ intersects one edge incident to u—see Fig. 4(ii).
Let (u, v) be the intersected edge incident to u. We claim that τ intersects another edge incident to v. Indeed, τ would otherwise be fully contained inside the union of the two triangles incident to (u, v), contradicting that σ

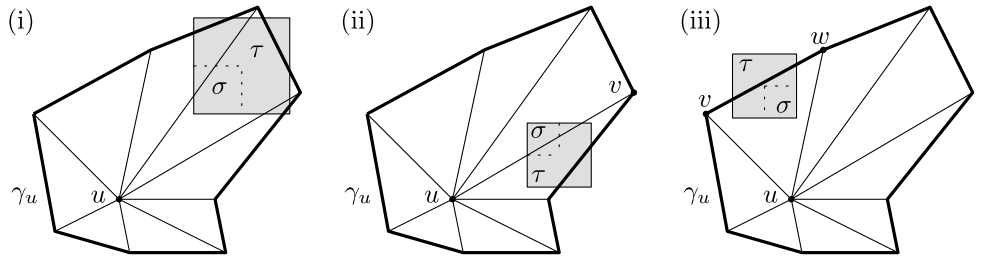

Fig. 4. The three cases in the proof of Lemma 2.5.

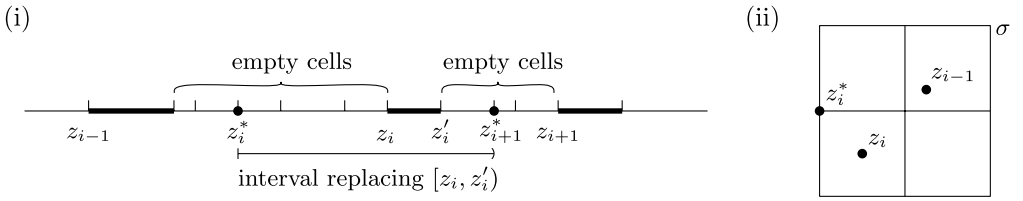

Fig. 5. Eliminating empty cells.

is final in $\mathcal{F}$. Hence, τ intersects another edge incident to v, as claimed. This means we can apply the argument for the first case to v, and conclude that σ is final in $star(v)$.

Case (iii): τ intersects zero edges incident to u—see Fig. 4(iii).

In this case σ does not intersect any edges at all and lies completely inside $\triangle$. Furthermore, τ intersects (v, w). We now apply the arguments given above with u and v changing roles, and find that σ is final in $star(v)$ or $star(w)$. □

On the basis of the above lemma, we can construct the linear quadtree as follows:

1. Compute an adjacency list for each vertex.
2. Scan the adjacency lists for all vertices. For each vertex u load its adjacency list into memory. Compute the minimal canonical square R containing $star(u)$. By recursively subdividing R, compute the final cells of $star(u)$ together with the triangles of $star(u)$ intersecting each final cell σ. Output each triangle with the key z of the Z-index interval $[z, z')$ that corresponds to σ.
3. Sort the triangles by key, removing duplicates.
4. Finally, build a (cache-oblivious) B-tree on the list of triangles with their keys.

Lemma 2.6. *The quadtree described above for a δ-fat triangulation with n edges can be constructed using $O(sort(n/\delta^2))$ I/Os.*

Proof. Step 1 can easily be carried out in $O(sort(n))$ I/O's (or faster if the triangulation is already given in a suitable way). As for Step 2, we observe that each star contains $O(1/\delta)$ triangles. Therefore, by Lemma 2.4, it contains $O(1/\delta^2)$ final cells, with $O(1/\delta^3)$ edge-cell intersections in total. By our assumption that $M \geqslant c/\delta^3$, for a large enough constant c, all computations for a star can be carried out in main memory. In total, again by Lemma 2.4, $O(n/\delta^2)$ cell-triangle intersections are computed (some of which may be computed twice), and so Step 2 takes $O(scan(n/\delta^2))$ I/O's in total. Step 3 then takes $O(sort(n/\delta^2))$ I/O's, after which Step 3 needs $O(scan(n/\delta^2))$ I/O's. □

Eliminating superfluous cells. The I/O-complexity of the construction and the storage requirements can be reduced with an easy optimisation in practice. The idea is to merge all final cells that lie properly inside triangles with their successors or predecessors in the Z-order. Consider a sequence of empty cells in between two non-empty cells $[z_{i-1}, z'_{i-1})$ and $[z_i, z'_i)$, as in Fig. 5(i). Then some of these empty cells will be merged with $[z_{i-1}, z'_{i-1})$ while the rest will be merged with $[z_i, z'_i)$. Which cells will be merged with $[z_{i-1}, z'_{i-1})$ is determined by a splitting value z_i^*, as defined below. Thus the interval $[z_i, z'_i)$ is effectively replaced by the interval $[z_i^*, z_{i+1}^*)$—see Fig. 5(i).

To implement this idea, we modify the algorithm as follows. In Step 2, we do not output so-called *empty* cells: cells that lie properly inside triangles. Instead we only output triangle-key pairs for triangle-cell intersections such that an edge of the triangle intersects the cell. We sort these triangle-key pairs, and then scan them, replacing their keys as follows. Let $z_1, \ldots, z_k$ be the different keys stored (each of them could be associated with several triangles). We leave key z_1 as it is, but replace each key z_i for $i > 1$ by a key z_i^* defined as follows. Consider the keys z_{i-1} and z_i. We identify the most significant bit that differs between z_{i-1} and z_i, and let z_i^* be the lowest Z-index in $[z_{i-1}, z_i]$ for which this bit has value 1. Replacing each z_i by z_i^* in this manner gives us the desired change: all triangles originally stored with key z_i will now be stored with

z_i^*, and the corresponding Z-index interval (implicitly) changes from $[z_i, z_i')$ to $[z_i^*, z_{i+1}^*)$. (Here we define z_{k+1}^* to be equal to z_k', which is in fact the Z-index of the point $(1, 1)$.)

For this approach to be correct, we must ensure the following: if an empty cell contained in a triangle Δ is merged with some non-empty cell σ, then Δ was already associated with σ. To be able to prove this, we need the following lemma.

Lemma 2.7. *For any $1 \leqslant i \leqslant k$, we have* (i) $[z_i, z_i') \subseteq [z_i^*, z_{i+1}^*)$, *and* (ii) *the region corresponding to $[z_i, z_{i+1}^*)$ and the region corresponding to $[z_{i+1}^*, z_{i+1}]$ are simply connected.*

Proof. The definition of z_i^* corresponds to the following. Consider the smallest canonical square σ containing both z_{i-1} and z_i, and assume that z_{i-1} lies in one of the two north quadrants of σ while z_i lies in one of the south quadrants, as in Fig. 5(ii). (The case where this is not the case, but instead they lie in the west and east quadrants, respectively, is similar.) Then z_i^* corresponds to the top left corner of the south-west quadrant of σ. Since $[z_{i-1}, z_{i-1}')$ is contained in one of the two north quadrants, this implies that $z_i^* \geqslant z_{i-1}'$. Thus, we have $z_{i-1}' \leqslant z_i^* \leqslant z_i$ for all i, which implies (i).

Consider the region R in the unit square corresponding to the Z-index interval $[z_i, z_{i+1}^*)$. Let N be the union of the two north quadrants of σ. Note that $R \subseteq N$. The definition of Z-index implies that all points q to the south-east of a point p have $Z(q) > Z(p)$. Hence, if a point p lies in R, then any point $q \in N$ to the south-east of p also lies in R. It follows that R is simply connected. A similar argument shows that the region corresponding to $[z_{i+1}^*, z_{i+1}]$ is simply connected, thus proving (ii). (Note that canonical squares are closed to the top and left, and open to the bottom and right. Hence, z_{i+1}^* belongs to the bottom quadrants of σ.) □

Now we can argue the correctness of our approach.

Lemma 2.7(i) states that original non-empty cells can only grow. What remains to show is that if an empty cell σ' contained in a triangle Δ is merged with some non-empty cell σ, then Δ was already associated with σ. To see this, consider an empty cell σ' with Z-index interval $[z, z')$, and suppose that $[z, z')$ lies in between z_i and z_{i+1}. Note that $[z, z')$ cannot contain z_{i+1}^* in its interior—see Fig. 5(ii)—so $[z, z')$ is either added completely to $[z_i, z_i')$ or it is added completely to $[z_{i+1}, z_{i+1}')$. Assume that the former is the case (the proof for the other case is analogous). We need to show that Δ is associated with the cell σ with Z-index interval $[z_i, z_i')$. By Lemma 2.7, the region corresponding to $[z_i, z_{i+1}^*)$ is simply connected. Therefore, from any point in σ' it is possible to draw a curve to a point p in σ such that this curve lies inside the region corresponding to $[z_i', z_{i+1}^*)$. Because this curve only traverses empty cells, it does not cross any edges and therefore it lies entirely in Δ. Hence p lies in Δ and p lies in σ, so Δ must have been associated with σ already, as claimed.

This establishes the correctness of our modified approach.

Updates. We support the following operations: inserting a vertex (and its incident edges), moving a vertex, deleting a vertex (and its incident edges, and re-triangulating the resulting hole in the triangulation), and flipping an edge. We assume that throughout the sequence of updates, the triangulation remains δ-fat.

By Lemma 2.5, all leaf cells that intersect a triangle $\triangle = (u, v, w)$ can be computed from the quadtrees of $star(u)$, $star(v)$ and $star(w)$. Since the size of $star(u)$, $star(v)$ and $star(w)$ is $O(1/\delta)$, by Lemma 2.4 the total number of cells that intersect $\triangle$ is $O(1/\delta^3)$. Since each of the supported operations changes only $O(1/\delta)$ triangles, we can compute the structure of the quadtree locally in the area of the update, and determine what the changes entail for the data stored on disk. All changes can thus be made in $O(\frac{1}{\delta^4}\log_B(n/\delta))$ I/Os when a normal B-tree is used, and in $O(\frac{1}{\delta^4}(\log_B(n/\delta) + \frac{1}{B}\log^2(n/\delta)))$ I/Os when a cache-oblivious B-tree is used [6,8].

2.4. Query operations

Next we show that our structure allows for efficient map overlay, point location, and range searching.

Lemma 2.8. *The linear quadtree for δ-fat triangulations as described above supports map overlay in $O(scan(n/\delta^2))$ I/Os.*

Proof. Our linear quadtree for a given triangulation is essentially a sorted list (with a B-tree built on top) whose elements are pairs $(\sigma, \triangle)$ such that $\triangle$ is a triangle intersecting the quadtree cell σ, where the ordering in the list is along the Z-order curve. More precisely, if we have an intersecting pair $(\sigma, \triangle)$ and the Z-order-interval of σ is $[z, z')$, then the triangle $\triangle$ will be stored in the list as the pair $(\text{key}(\sigma), \triangle)$, where $\text{key}(\sigma) = z$; the list is then sorted on these keys (along the Z-order curve). Note that when empty cells have been merged with non-empty cells as described in Section 2.3, keys may now represent regions that actually consist of multiple cells, but this does not affect the analysis given below.

To overlay two triangulations $\mathcal{F}_1$ and $\mathcal{F}_2$, each stored in a star-quadtree, we scan the two quadtrees (that is, sorted lists) simultaneously in Z-order, at any point keeping in memory the triangles stored with the last key read from the first list and those stored with the last key read from the second list. Starting from the beginning of the lists, we thus repeat the following until both lists have been read completely:

We select the list with the smallest unread key, read the next pair $(k, \triangle)$ from that list, and then read all other pairs $(k, \triangle')$ from that list having the same key k; this gives us all triangles intersecting the cell σ with $\text{key}(\sigma) = k$. For each

of these triangles, we check all the triangles associated with the current key in the other list—these are already stored in memory—and we report all intersecting pairs.

To see that this procedure computes all intersections, let $\triangle_1 \in \mathcal{F}_1$ and $\triangle_2 \in \mathcal{F}_2$ be two triangles that intersect. That means that there is a point p that lies inside each of them. Let z_1 and z_1' be the consecutive keys stored in the first star-quadtree such that $z_1 \leqslant Z(p) < z_1'$, and let z_2 and z_2' be the consecutive keys stored in the second star-quadtree such that $z_2 \leqslant Z(p) < z_2'$ (assuming each star-quadtree is concluded by a sentinel key 1, the keys z_1, z_1', z_2 and z_2' must exist). Since $\triangle_i$ $(i \in \{1,2\})$ intersects the Z-index interval $[z_i, z_i')$, it is stored with key z_i in the star-quadtree for $\mathcal{F}_i$. Note that $z_2 < z_1'$ and $z_1 < z_2'$, because both $[z_1, z_1')$ and $[z_2, z_2')$ contain $Z(p)$.

If $z_1 \leqslant z_2$, then, because $z_2 < z_1'$, the key z_1 will still be the last key read from the first quadtree when z_2 is reached in the second quadtree. If $z_2 < z_1$, then, because $z_1 < z_2'$, the key z_2 will still be the last key read from the second quadtree when z_1 is reached in the first quadtree. In either case the triangles with key z_1 are checked for intersections with the triangles with key z_2, and the intersection of $\triangle_1$ and $\triangle_2$ will be detected.

To avoid reporting a pair of intersecting triangles $\triangle_1$ and $\triangle_2$ more than once, we compute, every time such an intersecting pair is found, the point $p \in \triangle_1 \cap \triangle_2$ with lowest Z-index $Z(p)$. We only report the intersection of $\triangle_1$ and $\triangle_2$ when the last keys z_1 and z_2 read from the star-quadtrees satisfy $z_1 \leqslant Z(p)$ and $z_2 \leqslant Z(p)$; otherwise the analysis given above shows that the intersection must have been reported before.

Each cell intersects only $O(1/\delta)$ triangles by Lemma 2.4, so we never need to keep more than that many triangles in memory. Hence, the I/O-complexity of the algorithm is given by the time needed to scan the input and the time to write the output. For the former we need $O(scan(n/\delta^2))$ I/O's, also by Lemma 2.4. For the latter we need $O(scan(n/\delta))$ I/O's, since two δ-fat triangulations can have only $O(n/\delta)$ pairs of intersecting triangles. (This immediately follows from the fact that a δ-fat triangulation has density $O(1/\delta)$ [13].) □

Lemma 2.9. *The linear quadtree for δ-fat triangulations as described above supports point location in $O(scan(1/\delta) + \log_B(n/\delta))$ I/Os, where n is the number of points in the triangulation. Batched point location for k points can be done in $O(scan(n/\delta^2) + sort(k))$ I/Os.*

Proof. To perform point location with a point p, we simply compute the Z-index $Z(p)$ of p and search the B-tree for the triangles with the highest key less than or equal to $Z(p)$. We find the first of these triangles in $O(\log_B(n/\delta))$ I/Os and retrieve the others in $O(scan(1/\delta))$ I/Os—this is the list of triangles that intersect the cell σ that contains p—and we check which triangle contains p.

To perform batched point location, we sort the set of query points P by Z-index, and scan the leaves of the B-tree and P in parallel (similar to the overlay operation as described above) to find for each point p in which cell σ it lies, and within σ, in which triangle of the list of triangles stored for σ. Since the list of triangles of each cell σ has size only $O(1/\delta)$ it fits in memory, so that it needs to be read from disk only once even if multiple query points p lie in σ. Thus the full scan takes $O(sort(|P|) + scan(|P|) + scan(n/\delta^2)) = O(sort(k) + scan(n/\delta^2))$ I/Os. □

To prove the bound for range searching, we need the following lemma. Recall that Q_ε denotes the Minkowski sum of a range Q and a disk of radius $\varepsilon \cdot \mathrm{diam}(Q)$.

Lemma 2.10. *Given any constant-complexity range Q and a parameter $1 \geqslant \varepsilon > 0$, we can compute, in $O(1/\varepsilon)$ time and I/Os, a collection Ψ_Q of $O(1/\varepsilon)$ disjoint canonical squares such that $Q \subset \bigcup\{\sigma\colon \sigma \in \Psi_Q\} \subset Q_\varepsilon$.*

Proof. Let w^* be the width of the largest canonical square whose diameter is less than $\varepsilon \cdot \mathrm{diam}(Q)$. Note that $w^* \geqslant \varepsilon \cdot \mathrm{diam}(Q)/2\sqrt{2}$. Now start subdividing the unit square into subsquares in a recursive manner, stopping as soon as a square has width w^* or it does not intersect ∂Q. Let Ψ_Q be the set of squares obtained in this manner, removing the ones that do not intersect Q. It is easy to see that $Q \subset \bigcup\{\sigma\colon \sigma \in \Psi_Q\} \subset Q_\varepsilon$.

To prove the bound on the size of Ψ_Q, we charge each $\sigma \in \Psi_Q$ to its parent. (Note that the parent will not be a square in Ψ_Q.) Let Γ denote the collection of these parents. Obviously, every square in Γ gets charged at most four times. By construction, every square in Γ intersects ∂Q. We now use the following fact proved by Haverkort et al. [19]:

> Partition the unit square into a regular grid whose cells have width w. Then ∂Q intersects $O(1 + \frac{\mathrm{diam}(Q)}{w})$ cells of the grid.

Define $w_i = 1/2^i$. Note that each square in Γ has width w_i for some $i \in \{0, 1, \ldots, -\log w^*\}$. In fact, any square in Γ has width at most $\sqrt{2} \cdot \mathrm{diam}(Q)$—this is because any square in Ψ_Q has diameter at most $\mathrm{diam}(Q)$ and width at most $\mathrm{diam}(Q)/\sqrt{2}$—and so squares in Γ have width w_i for some $i \in \{i^*, \ldots, -\log w^*\}$ where $i^* \geqslant \log(\frac{1}{\sqrt{2}\cdot \mathrm{diam}(Q)})$.

By the fact mentioned above, there are at most $O(1 + \frac{\mathrm{diam}(Q)}{w_i})$ squares in Γ that have width w_i. Hence, the total number of squares in Γ is bounded by

$$\sum_{i=i^*}^{-\log w^*} O\left(1 + \frac{\mathrm{diam}(Q)}{w_i}\right) = O\left(-\log w^* - i^* + \mathrm{diam}(Q) \cdot \sum_{i=0}^{-\log w^*} 2^i\right)$$

$$
\begin{aligned}
&= O\left(\log\left(\frac{2\sqrt{2}}{\varepsilon\cdot\operatorname{diam}(Q)}\right) - \log\left(\frac{1}{\sqrt{2}\cdot\operatorname{diam}(Q)}\right) + \operatorname{diam}(Q)/w^*\right)\\
&= O\left(\log(4/\varepsilon) + \operatorname{diam}(Q)/w^*\right)\\
&= O(1/\varepsilon),
\end{aligned}
$$

which proves the claimed bound on the size of Ψ_Q.

Note that Ψ_Q is computed top-down by splitting $O(|\Psi_Q|)$ cells. By the definition of a constant-complexity query range (see the beginning of Section 2, the decision whether to split a cell can be taken in constant time. Therefore Ψ_Q can be computed in $O(|\Psi_Q|) = O(1/\varepsilon)$ time and I/Os. □

Using Lemma 2.10, we can now prove our result on range searching.

Lemma 2.11. *The linear quadtree for δ-fat triangulations as described above can be used to report, for any constant-complexity query range Q and for any constant $\varepsilon > 0$, the triangles that intersect Q in $O(\frac{1}{\varepsilon}scan(1/\delta) + \frac{1}{\varepsilon}\log_B(n/\delta) + scan(k_\varepsilon/\delta^2))$ I/Os, where k_ε is the number of triangles that lie at distance at most $\varepsilon\cdot\operatorname{diam}(Q)$ from Q.*

Proof. To answer a query, we compute the collection Ψ_Q of Lemma 2.10, and we perform a query with each square $\sigma\in\Psi_Q$. Since each square $\sigma\in\Psi_Q$ is canonical, it corresponds to a single interval on the Z-order curve. Hence, we can perform the query by searching with the starting point of this interval in the B-tree and then scan from there until the interval endpoint; during the scan we report all triangles that intersect the query range and have not been reported already. Note that it is easy to check whether a triangle has been reported before: given the triangle Δ, the Z-index key z with which it is stored, and the query range Q, one can compute whether there is a point p with Z-index less than z that also lies inside Δ and inside Q. If not, the triangle Δ is reported. Otherwise Δ must have been reported already when the Z-order scan reached the interval containing p. The total number of I/O's is thus $O(\frac{1}{\varepsilon}\log_B(n/\delta))$ plus the number of I/O's needed for the scans.

It remains to analyze how may triangles we encounter over all these scans. (Since triangles can be stored multiple times, we may encounter a triangle more than once.) To this end we consider the cells in our quadtree that lie within the squares in Ψ_Q; these are exactly the cells whose contents we scan. If the parent of such a cell lies completely inside Q_ε then we can charge the triangle-cell intersection to vertices and edges that intersect Q_ε, as in Lemma 2.4; their contents can therefore be scanned in $O(scan(k_\varepsilon/\delta^2))$ I/Os. Otherwise the cell has diameter $\Omega(\varepsilon\cdot\operatorname{diam}(Q))$, and as in the proof of Lemma 2.10, we can bound the number of such cells by $O(1/\varepsilon)$; hence, scanning their contents takes $O(\frac{1}{\varepsilon}scan(1/\delta))$ I/O's. □

3. Low-density subdivisions: the guard-quadtree

In this section we describe our guard-quadtree for storing planar low-density subdivisions. The *density* of a set S of objects in the plane is the smallest number λ such that the following holds: any disk D is intersected by at most λ objects $o\in S$ such that $\operatorname{diam}(o)\geqslant\operatorname{diam}(D)$ [12]. We say that a planar subdivision $\mathcal{F}$ has density λ if its edge set has density λ. In other words, any disk D is intersected by at most λ edges whose length is at least the diameter of D. We can apply a uniform scaling and translation to the plane without changing the density of the input subdivision $\mathcal{F}$. Therefore, we assume without loss of generality that $\mathcal{F}$ is contained in the unit square $[0,1)^2$. In this section we will prove the following result.

Theorem 3.1. *Let $\mathcal{F}$ be a subdivision of density λ and with n edges. Assume that the memory size M is at least $c\lambda$, for a large enough constant c. We can construct, in $O(sort(n)\log\lambda)$ I/Os a linear compressed quadtree for $\mathcal{F}$ that stores $O(n)$ cell-edge intersections. We call this quadtree the* guard-quadtree *for $\mathcal{F}$. With the guard-quadtree we can perform the following operations*:

(i) **Map overlay:** *If we have two subdivisions of density λ with n edges in total, both stored in a guard-quadtree, then we can find all pairs of intersecting edges in $O(scan(n+k))$ I/Os, where $k = O(\lambda n)$ is the complexity of the output.*

(ii) **(Batched) point location:** *For any query point p we can find the face of $\mathcal{F}$ that contains p in $O(scan(\lambda)+\log_B n)$ I/Os, and for any set P of k query points we can find the face of $\mathcal{F}$ that contains each point p in $O(scan(n)+sort(k))$ I/Os.*

(iii) **Range searching:** *For any constant-complexity query range Q and for any constant $\varepsilon > 0$ we can find the edges of $\mathcal{F}$ that intersect Q in $O(\frac{1}{\varepsilon}scan(\lambda)+\frac{1}{\varepsilon}(\log_B n)+scan(k_\varepsilon))$ I/Os, where k_ε is the number of edges that lie at distance at most $\varepsilon\cdot\operatorname{diam}(Q)$ from Q.*

All algorithms can be implemented cache-obliviously.

The results, except point location, also hold for sets of n segments that do not form a proper subdivision (even if they intersect), as long as the density of the set is λ.

Below we explain our data structure, and how to construct it. The query algorithms are the same as described in the previous section, although for range searching, a slightly different analysis is needed. Note that the results of this section can be used for δ-fat triangulations since any set of disjoint δ-fat triangles in the plane has density $O(1/\delta)$ [13]. However, the solution from the previous section is simpler and dynamic.

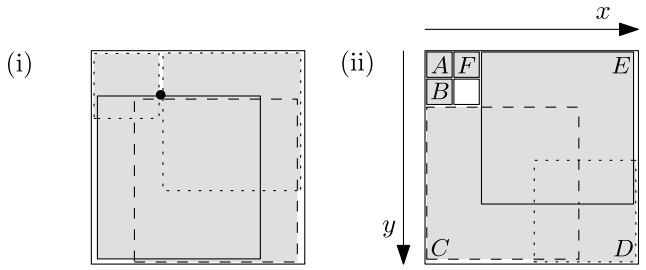

Fig. 6. Covering cells by squares. (The squares are drawn slightly smaller than their actual sizes.)

3.1. The compressed quadtree subdivision for low-density maps

Let $\mathcal{F}$ be a subdivision of the unit square with n edges and of density λ. In general it is impossible to construct a standard quadtree on $\mathcal{F}$ consisting of a linear number of cells that are each intersected by a constant number of edges. Indeed, in a general subdivision of the unit square there can be many vertices arbitrarily close together, even if the subdivision has constant density. To overcome this problem we shall use a so-called *compressed quadtree*.

Let $\mathcal{G}$ be the set of vertices of the axis-parallel bounding boxes of the edges of $\mathcal{F}$; we call the points in $\mathcal{G}$ *guards*. This set has the following nice property:

Lemma 3.2. *(See [12].) If S is a set of objects with density λ, then any square σ intersects at most 4λ objects from S that do not have a bounding-box corner in the interior of σ.*

This lemma implies that if we build a quadtree such that each cell contains only few guards, each cell will intersect only few edges. For simplicity we assume that a guard of one edge and a guard of another edge only coincide if the guard is a shared endpoint of the two segments.[1] Since at most λ edges share a common endpoint, this means that at most λ guards coincide in a single point. The idea is now to base our quadtree for $\mathcal{F}$ on the following stopping rule.

Stopping rule for low-density subdivisions: Stop splitting when the cell σ does not have a guard in its interior or when all guards in its interior coincide.

Consider the quadtree obtained from this stopping rule. Its cells intersect $O(\lambda)$ edges each, but the number of cells cannot be bounded. Hence, we compress the quadtree [21] by building it as follows. We recursively subdivide each canonical square σ that contains at least two points from $\mathcal{G}$ that do not coincide into *five* regions. Let o be the smallest canonical square that contains all points of $\sigma \cap \mathcal{G}$. The first region is the *donut* $\sigma \setminus o$. The remaining four regions are the four quadrants of o. Note that the first region does not contain any points of $\mathcal{G}$, so it is not subdivided further. When $o = \sigma$, the first region is skipped; when o is smaller than σ, we call $\sigma \setminus o$ a *proper donut*.

Lemma 3.3. *Let $\mathcal{F}$ be a subdivision $\mathcal{F}$ of the unit square with n edges and of density λ. Then a compressed quadtree subdivision based on the stopping rule defined above has $O(n)$ cells, and each cell of the quadtree subdivision is intersected by at most 24λ edges of $\mathcal{F}$.*

Proof. Note that each internal node of the compressed quadtree has points from $\mathcal{G}$ in at least two of its subtrees, hence, the total number of nodes is $O(|\mathcal{G}|) = O(n)$.

If a cell σ is a square, all guards in σ coincide in one point p. Hence σ can be covered by four (not necessarily disjoint) squares that lie completely in σ and do not have any guards in their interior—see Fig. 6(i). By Lemma 3.2, we conclude that σ is intersected by at most 16λ edges.

If σ is a proper donut, it is empty of guards. Moreover, it can be covered by six squares that do not contain any guards in their interior. This can be seen as follows. Let W be the width of σ, let w be the width of the hole, let d_x be the distance between the hole and the left edge of σ, and let d_y be the distance between the hole and right edge of σ. We assume $d_x \geqslant d_y$ and $d_x + w \leqslant \frac{1}{2}W$ (other cases are symmetric). We now define six squares, of which the corners have the following coordinates relative to the upper-left corner of σ:

Fig. 6(ii) gives an example. Because the hole is a canonical square inside σ, an integral number of canonical squares of width w fits between the hole and the outer boundary of σ. Thus either $d_x = 0$ or $d_x \geqslant w$, and either $d_y = 0$ or $d_y \geqslant w$. The reader may now verify the following: all squares A to F lie inside σ; none of these squares intersects the hole; A, B and C together cover the area inside σ that lies to the left of the hole; C and D cover the area inside σ that lies below the hole;

[1] In fact, this can be ensured by choosing the guards in a slightly different manner. Namely, we can replace a guard g of an edge e that is not one of e's endpoints by two suitably chosen other guards very close to g. This can be done such that the new guards do not coincide with any other guards.

square	upper-left corner	lower-right corner
A	$(0,0)$	(d_x, d_x)
B	$(0, d_y)$	$(d_x, d_y + d_x)$
C	$(0, d_y + w)$	$(W - d_y - w, W)$
D	$(d_y + w, d_y + w)$	(W, W)
E	$(d_x + w, 0)$	$(W, W - d_x - w)$
F	$(d_x, 0)$	$(d_x + d_y, d_y)$

D and E cover the area inside σ that lies to the right of the hole, and A, E and F together cover the area inside σ that lies above the hole. Thus each donut can be covered by six squares that do not contain any guards in their interior—and hence at most 24λ edges intersect it. □

Unfortunately, the simple stopping rule given above does not quite suffice. First of all, the total number of cell-edge intersections is $O(n\lambda)$. Although this is not bad when λ is small, we would prefer it to be $O(n)$. A second problem is that with the compressed quadtree subdivision defined above we cannot give output-sensitive bounds on the number of I/Os taken by range queries. This is caused by the fact that some of the bounding-box vertices of an edge may lie far from the edge. As a result, it is possible that locally many small cells are made to separate the guards of edges that lie far from those guards. A range query may then need to scan many cells that cannot be charged to edges that lie inside or close to the query range.

To overcome these problems, the idea is to proceed as follows.

First of all, we stop subdividing as soon as the number of guards inside a cell is at most λ. (Here coinciding guards are counted with multiplicity.) This will reduce the number of cells to $O(n/\lambda)$, while increasing the number of edges per cell only by an additional term λ (so that it is still $O(\lambda)$) [12].

Second, when deciding whether to subdivide a cell, we would like to only take into account guards that are relevant—that is, guards whose corresponding object intersects the cell.

Implementing these ideas is not so simple, however. For instance, we do not know the value of λ. More importantly, if we want to compute the structure in an I/O-efficient manner, we cannot afford a top-down approach that simply splits cells in a recursive manner until the stopping criterion is met. Hence, we take a different route: we first compute the compressed quadtree with the simple stopping rule given above, and then we reduce the number of cells by merging them in a suitable manner. How all of this can be done I/O-efficiently will be described below. But first we explain how the compressed quadtree will be stored.

3.2. Storing the compressed quadtree subdivision and the low-density map

We store the cell-edge intersections of the compressed quadtree subdivision in a list sorted by the Z-order of the cells, indexed by a (cache-oblivious) B-tree. The only difference with the previous section is that we now have to deal with donuts as well as square cells. Recall that a canonical square—a square that can be obtained from the unit square by a recursive partition into quadrants—corresponds to an interval on the Z-order curve. For a donut this is not true. However, a donut corresponds to at most two such intervals, because a donut is the set-theoretic difference of two canonical squares. Thus, the solution of the previous section can be applied if we represent each donut by two intervals $[z_1, z_1')$ and $[z_2, z_2')$; edges intersecting the first part of the donut are stored with key z_1, and edges intersecting the second part are stored with key z_2, edges intersecting both parts are stored twice. We will call the regions covered by $[z_1, z_1')$ and $[z_2, z_2')$ *half-donuts*. As described in Section 2.3, we merge cells that do not intersect any edge with their immediate successors or predecessors in the Z-order (as before, in such a way that merged cells are connected). We call the resulting structure—the B-tree on the cell-edge intersections whose keys define a compressed quadtree subdivision—a *linear compressed quadtree.*

3.3. Building the compressed quadtree I/O-efficiently

We construct the leaf cells of the compressed quadtree, or rather, the corresponding subdivision of the Z-order curve, as follows. We sort the set of guards $\mathcal{G}$ into Z-order, and scan the sorted guards. For each pair of consecutive guards, say p and q, that do not coincide, we construct their lowest common ancestor $\mathrm{lca}(p,q)$ (the smallest canonical square that contains both of them) by examining the longest common prefix of the bit strings representing $z(p)$ and $z(q)$. We output the five Z-indices that bound and separate the Z-order intervals of the four children of $\mathrm{lca}(p,q)$. To complete the subdivision of the Z-order curve, we sort the output into a list $\mathcal{L}$ by Z-order and remove duplicates.

Lemma 3.4. *The above algorithm generates, in $O(\mathrm{sort}(n))$ I/Os, a subdivision of the Z-order curve that corresponds to a compressed quadtree on $\mathcal{G}$.*

Proof. The definition of the compressed quadtree implies that a canonical square σ is subdivided into its four quadrants if and only if it contains at least two guards that lie in different quadrants of σ. If such a pair exists, at least one such pair (p,q) appears as a pair of consecutive guards in the Z-order, since the Z-order curve visits all guards in σ consecutively.

The above algorithm generates the boundaries of the four quadrants of σ when the pair (p, q) is encountered. Furthermore, whenever the above algorithm generates the boundaries of the four quadrants of a cell σ, there is a pair (p, q) that has σ as their lowest common ancestor and, therefore, lie in different quadrants of σ.

It remains to check that the boundaries of donut cells are generated correctly. The Z-indices that define the outer square of a donut $\sigma \setminus o$ appear when a pair of guards is processed that consists of a guard in the hole o and a guard in a sibling of σ. The Z-indices that define the hole o appear when a pair of guards is processed that lie in different quadrants of the inner square. □

Although the above algorithm runs in $O(sort(n))$ I/Os, it does generate multiple copies of cell boundaries, so that duplicates need to be sorted and removed. With a relatively simple modification, it is possible to avoid generating duplicate cells, and thus avoid the need for a sorting pass—see Appendix A for the details.

Merging cells. As mentioned above, we need to merge cells that are too small into bigger cells to reduce the size of the data structure and to enable efficient range queries. We will now make precise when a cell is considered to be too small, and how we can merge such cells.

We define a guard p to be *relevant* to a cell σ if p lies in σ and σ intersects the edge of which p is a guard. The *relevance size* of a guard p is the size of the smallest canonical square σ such that p is relevant to σ. (If a guard is the endpoint of an edge, its relevance size is zero.) We now fix a threshold λ^*; we will discuss how to choose λ^* later. For any two regions σ and τ, where $\sigma \subseteq \tau$, we say that σ is *τ-relevant* if it contains at least λ^* guards that are relevant to τ. Note that if σ is τ-relevant, and ϕ contains τ, then σ is also ϕ-relevant. We call a cell σ *self-relevant* if it is σ-relevant.

The idea of our merging algorithm is that we try to merge cells that are not self-relevant. Given a compressed quadtree, a natural attempt to do so would traverse the quadtree top-down in depth-first order, replacing every subtree of which the root is not self-relevant by a single leaf. However, there are two problems with this approach. First, we do not have the complete compressed quadtree available; we only have the sorted list of Z-indices defining the boundaries of the Z-order intervals defining the leaf cells. Second, this approach could still leave cells that have a long chain of ancestors, all of whose siblings intersect only very few guards and edges. In such cases, the siblings should be merged into a donut shape.

Therefore we do the merging bottom-up. The idea is that we simulate a post-order traversal of a hypothetical quadtree on top of the leaf cells. To this end we scan the list of Z-indices that bound the cells and collect consecutive intervals that represent squares and donuts that have a common parent. When we find a set of squares and donuts that have a common parent we decide whether they should be merged into a single leaf square or a single donut. Squares and donuts of which not all siblings have been found yet, are kept on a stack. A minor complication is that the half-donuts make it hard to say if cells would actually have a common parent if the quadtree had been constructed top-down. Fortunately, for our purposes we may simply consider all parents that *could* exist: we will consider to merge cells whenever they cover a canonical square together. To decide if cells should indeed be merged, we need to know how many relevant guards the resulting cell would contain. Therefore, while we scan the Z-order interval boundaries, we also need to consider the guards and maintain information about the guards in the cells on the stack. Next we make these ideas precise.

Our algorithm takes as input the Z-indices that define the compressed quadtree subdivision computed before, and the set of guards, all given in one list sorted by Z-order. Note that each guard in this list is mentioned once for every edge of which it is a bounding-box corner. The output of the algorithm will be another set of Z-indices that induce a compressed quadtree subdivision. Furthermore, the algorithm will produce a tree $\mathcal{T}$ of quadtree regions—this could be omitted from an actual implementation, but the tree construction will be helpful in proving properties of the compressed quadtree subdivision that is induced by the Z-indices output.

The algorithm will maintain an I/O-efficient stack to keep track of cells that are eligible for merging. Initially the stack is empty. In the course of the algorithm, regions will be pushed on the stack. Each region is of one of the following types:

- a single square;
- a half-donut;
- a donut set, that is, a square τ that consists of one or two half-donuts that together form a donut and a square that forms the hole—the hole must be τ-relevant.

On the stack each square and each half-donut σ is stored with a list of guards in σ sorted by increasing relevance size, truncated to the first λ^* guards. For a donut set, we store such a list separately for each of the (at most) three constituents of the donut set.

The merging algorithm now proceeds as follows. We scan, one by one, the cells of the input quadtree subdivision with the guards contained in them, and push each cell onto the stack together with its first λ^* guards in order of increasing relevance size.

After every addition to the stack, we check the topmost entries of the stack to see if a square region has now been read completely and some or all cells in it could be merged. To do so, we inspect the topmost k regions of the stack, first for $k = 2$. If their union is equal to the smallest canonical square τ that contains both regions, then they are a candidate for merging. If the topmost region on the stack does not end at the same Z-index as τ, then we know some siblings still need

to be read and we proceed by scanning the next cell from the input subdivision. If the first two regions turn out not to be a candidate for merging, we check again with $k = 3$, and finally with $k = 4$. (Below we will explain why it is not necessary to check for $k > 4$.)

When a $k \in \{2, 3, 4\}$ is found such that the topmost k entries on the stack are a candidate for merging, we pop them from the stack. Some of these k entries may actually be donut sets, and thus we get a set S of at most $3k \leqslant 12$ squares and half-donuts. Now we distinguish three cases:

1. If at least two of the squares and half-donuts in S are τ-relevant, we output the Z-indices that separate the squares and half-donuts of S from each other and we push τ on the stack. This is reflected in $\mathcal{T}$ as follows: we construct a node for τ and a node for each element of S for which no node was constructed yet; the nodes for the elements of S are given as children to the node for τ.
2. Otherwise, if S contains exactly one square σ (*not* a non-square half-donut) that is τ-relevant, we remove the squares and half-donuts in S that precede σ in Z-order from S, and replace them by one half-donut σ_{prev} which is the union of the removed squares and half-donuts; we also remove the squares and half-donuts in S that follow σ in Z-order from S, and replace them by one half-donut σ_{next}. If at least one of σ_{prev} and σ_{next} is τ-relevant, we proceed as in Case 1. Otherwise we push σ_{prev}, σ and σ_{next} on the stack as a donut set.
3. Otherwise we push τ on the stack.

Note that pushing τ, and, when applicable, σ_{prev} and σ_{next} on the stack involves constructing lists of guards: the lists of guards of the squares and half-donuts that were popped from the stack, are combined into lists for τ, σ_{prev}, and σ_{next}, truncated to the λ^* guards with the smallest relevance size. After treating the set S as described above, we check again if the topmost 2, 3 or 4 entries on the stack are a candidate for merging. When no candidates for merging are found, we resume scanning the input cells.

When the input has been read completely and all candidates for merging have been considered, the stack will hold the result of processing the unit square. If this is a donut set, we output the Z-indices that delimit the hole in the donut, otherwise nothing needs to be done. Finally we sort all Z-indices that were output by the algorithm into Z-order.

The above algorithm produces a compressed quadtree subdivision—in the form of a list of Z-indices that induce the subdivision—and the corresponding compressed quadtree $\mathcal{T}$. Let σ_ν be the region associated with a node ν of the quadtree, and let n_ν be the number of guards in σ_ν that are relevant to σ_ν. The quadtree $\mathcal{T}$ has the following property.

Lemma 3.5. *In the compressed quadtree $\mathcal{T}$ produced by the above algorithm, the following holds for any internal node ν: we have $n_\nu \geqslant \lambda^*$, and the number of leaf cells below ν is $O(n_\nu/\lambda^*)$. Each leaf cell σ intersects less than $24\lambda + 4\lambda^*$ edges.*

Proof. The above algorithm only checks if the topmost k entries of the stack can be merged for $k \in \{2, 3, 4\}$. We do not need to check for $k > 4$. This follows from the fact that the original subdivision is constructed by means of cuts where each square is cut into a donut (consisting of one or two Z-order intervals) and a square hole (cut into four quadrants). As soon as the last quadrant of the hole is completed, the four quadrants of the hole will be on top of the stack ($k = 4$) and they will be replaced by a single square region. The square will subsequently be combined with the surrounding half-donuts, in one or two steps, with $k = 2$ or $k = 3$.

Only when handling Case 1 are Z-indices output; more precisely we output the Z-indices that separate up to $3k \leqslant 12$ regions in a self-relevant square τ from each other. The Z-indices that bound τ are not output immediately, but they will be output later: τ is first put on the stack; when it is popped later, it may be put back on the stack (Case 2) without further processing, but eventually it will be popped and end up in a set S that is handled in Case 1, so that the Z-indices that separate τ from other cells will eventually be output.

Therefore, to analyse the number of cells produced, we only have to consider the cells that are associated with tree nodes while handling Case 1. By construction, every internal node μ in a subtree of $\mathcal{T}$ rooted at ν has at least two children whose corresponding regions are σ_μ-relevant, that is, these children contain at least λ^* guards that belong to edges intersecting σ_μ, and thus, σ_ν. It follows that the total number of nodes constructed in the subtree below ν is $O(n_\nu/\lambda^*)$, where n_ν is the number of relevant guards in σ_ν.

To bound the number of relevant guards in each cell, consider the leaf nodes constructed while handling Case 1. These correspond to cells σ that were formed in one of the following ways:

- σ was read from the original compressed quadtree subdivision. By Lemma 3.3, σ intersects at most 24λ edges.
- σ is a square or half-donut formed while handling Case 2 or 3 by merging up to four cells that are all contained in some square τ but are not τ-relevant. Thus, σ contains less than $4\lambda^*$ guards that are relevant to τ and, since σ lies in τ, the same bound holds for guards relevant to σ itself. The proof of Lemma 3.3 shows that σ intersects at most 24λ edges, apart from those that have a guard in σ.
- σ was formed while handling Case 3 by merging a τ-relevant half-donut ϕ, the other half of the donut (if it exists), and the hole into a single square τ. However, half-donuts from the original compressed quadtree subdivision do not contain guards and cannot be τ-relevant, and any half-donut formed by merging cells comes in a donut set with a square hole that is τ-relevant for any square τ that contains the donut. Thus, we would have had Case 1 or 2, not Case 3. □

A bottleneck in the above approach is that it involves $O(n)$ stack operations and each stack operation involves a list of λ^* guards. To speed up the approach described above, we first perform a rough pre-merging step to reduce the number of cells that need to be considered. In the pre-merging step, we run the algorithm described above with two modifications. First, instead of measuring relevance to the square τ that is currently being processed, we always measure relevance to the unit square—which implies that all guards in a cell are always relevant. Second, because of this, we do not need to keep lists of guards sorted by relevance size; it suffices to simply count the total number of guards in a region—this will speed up the stack operations. After the pre-merging step, we run the original merging algorithm described above on the compressed quadtree subdivision that results from the pre-merging step.

Lemma 3.6. *In $O(sort(n))$ I/Os we can compute the Z-indices that induce a compressed quadtree subdivision, corresponding to a compressed quadtree $\mathcal{T}$, such that the following holds for any internal node v of $\mathcal{T}$: we have $n_v \geqslant \lambda^*$, and the number of leaf cells below v is $O(n_v/\lambda^*)$. Each leaf cell σ intersects less than $24\lambda + 6\lambda^*$ edges.*

Proof. Following the proof of Lemma 3.5, the pre-merging step reduces the number of cells in the compressed quadtree subdivision to $O(n/\lambda^*)$, and each cell intersects less than $24\lambda + 4\lambda^*$ edges.

For the main merging step, the analysis given above still holds, except that we now have to take into account that cells in the input quadtree subdivision—also half-donuts—may contain up to $4\lambda^*$ guards. This affects the analysis of the third way in which a cell could be formed, where σ is formed while handling Case 3 by merging a τ-relevant half-donut ϕ with the other half-donut and the hole into a single square τ. Observe that in this case, since ϕ contains less than $4\lambda^*$-guards in total, and the other half-donut and the hole each contain less than λ^* τ-relevant guards (because otherwise we would have had Case 1, not Case 3), the resulting cell σ has less than $6\lambda^*$ guards whose edges may intersect σ.

To analyse the number of I/Os needed, observe that both the pre-merging and the final merging step consist of (1) scanning the input cells, (2) pushing them on the stack, and a number of merging and output operations each of which reduces the number of items on the stack by at least one, and (3) sorting the output. The number of stack operations is therefore bounded by the number of cells in the input quadtree subdivision. With each stack operation in the pre-merging step, we transfer $O(1)$ data; thus, the number of I/Os for the pre-merging step is $O(scan(n)+scan(O(n)\cdot O(1))+sort(n/\lambda^*))$. With each stack operation in the second merging step we transfer $O(\lambda^*)$ data (lists of guards sorted by relevant size); thus, the number of I/Os for the final merging step is $O(scan(n) + scan(O(n/\lambda^*)\cdot O(\lambda^*)) + sort(n/\lambda^*))$. This adds up to $O(scan(n) + sort(n/\lambda^*))$, which is never worse than $O(sort(n))$. □

Computing the intersection of $\mathcal{F}$ and the quadtree. Having constructed the compressed quadtree subdivision, we can now distribute the edges in $\mathcal{F}$ to the cells of the quadtree subdivision, or rather, to the corresponding Z-order intervals. We first describe a cache-aware algorithm, and then we argue that it can be made cache-oblivious.

We start by building a balanced tree $\mathcal{Z}$ on the subdivision of the Z-order curve as computed above; we make sure that the internal nodes of $\mathcal{Z}$ just above leaf level have degree roughly c_1M, and the internal nodes on higher levels have degree roughly c_2M/B, for sufficiently small constants c_1 and c_2. The root may have arbitrarily low degree. We initialize an output stream $\mathcal{I}$, to which we will write intersections between edges and cells.

Let S be the set of children of the root of $\mathcal{Z}$. Each of the nodes in S covers a certain interval of the Z-order curve, and together they form a subdivision of the Z-index interval covered by the root. We load each node of S into memory.

If the nodes in S are internal nodes, we assign an output stream to each of these nodes, and reserve a buffer of one block in memory for each of them. We now read the edges from the input one by one, and distribute each edge to the output streams of the nodes whose Z-order interval intersects the edge. Note that each edge may be copied to several streams. Once all edges have been read, we distribute the edges in each node's stream recursively into the subtree rooted at that node.

If the nodes in S are leaves, we do not make buffered output streams and recursive calls for them; instead we write the intersections between edges and leaf cells to $\mathcal{I}$.

Note that the algorithm is in effect a $\Theta(M/B)$-way distribution to distribute a sequence of edges to the intervals in the quadtree subdivision that intersect them.

Lemma 3.7. *We can compute all intersections between the $O(n)$ edges of $\mathcal{F}$ and the quadtree subdivision of Lemma* 3.6 *in $O((n'(\lambda+\lambda^*)/B)\log_{M/B}(n'/B))$ I/Os, where $n' = n/\lambda^*$.*

Proof. By Lemma 3.6, the total number of cell-edge intersections is $O(n'(\lambda+\lambda^*))$. There are $O(\log_{M/B}(n'/B))$ levels of recursion, and on each level, at most $O(n'(\lambda+\lambda^*))$ segments are distributed over at most $O(n'/M)$ output streams. This takes $O((n'(\lambda+\lambda^*)/B)\log_{M/B}(n'/B))$ I/Os in total. □

Note that computing whether an edge e intersects (the region corresponding to) a Z-order interval may require care, since such intervals do not need to correspond to a simple canonical square but may correspond to regions that have more complex shapes. However, the computation can be done in main memory and does not affect the I/O-bounds of our algorithm. (More precisely, to check whether an edge intersects the region corresponding to a Z-order interval we iteratively

generate its canonical squares in Z-order, and check each of them for intersection with e; after checking a canonical square it can be discarded immediately, so the amount of storage is constant and not related to the complexity of the shape of the area.)

As described above, the algorithm needs to know the value of M and B. We now argue that this distribution can be run cache-obliviously in a manner similar to lazy funnelsort [7]. Lazy funnelsort is based on *k-mergers*: a k-merger takes as input k sorted sequences and merges them into a sorted sequence. We will use k-mergers the other way around: using them to *distribute* edges from one sequence into k sequences. We will call these reverted mergers *k-distributors*.

In our setting, a k-distributor will be associated with a set of k Z-index intervals $Z_1, \ldots, Z_k$, whose union is a Z-index interval Z. A k-distributor is a data structure that will be used to distribute edges that intersect Z to k streams of edges intersecting $Z_1, \ldots, Z_k$, respectively. We assume that $\log k$ is a power of two; in other words, $\log\log k$ is an integer. When $k = 2$, the k-distributor is an empty data structure. A k-distributor D for $k > 2$ consists of a $\sqrt{k}$-distributor D_0, a set of buffers $B_1, \ldots, B_{\sqrt{k}}$ of size $k^{3/2}$ each, and a set of $\sqrt{k}$-distributors $D_1, \ldots, D_{\sqrt{k}}$. Each distributor D_i, for $1 \leqslant i \leqslant \sqrt{k}$, is associated with the buffer B_i (which serves as its input buffer) and the Z-index intervals Z_j with indices j such that $(i-1)\sqrt{k} < j \leqslant i\sqrt{k}$. The $\sqrt{k}$-distributor D_0 is used to distribute the input edges of D to the buffers $B_1, \ldots, B_{\sqrt{k}}$; the distributors $D_1, \ldots, D_{\sqrt{k}}$ are then used to distribute the edges from the buffers to the k output streams of D.

To distribute a set of edges S using a k-distributor D, we push the edges of S into D one by one. When an edge e is pushed into a 2-distributor, it is written to the output stream(s) or buffer(s) for which e intersects the corresponding Z-index interval. When an edge e is pushed into a k-distributor D with $k > 2$, we push it into the $\sqrt{k}$-distributor D_0 recursively. As a result, D_0 may send some edges to its output buffers $B_1, \ldots, B_{\sqrt{k}}$, which may run full. Whenever a buffer B_i runs full, we empty B_i by pushing its contents into D_i before we push more edges into D_0.

The complete distribution process is now done as follows. Let z be the number of cells of the compressed quadtree subdivision produced by the algorithm of Lemma 3.6. We divide these cells, or in order words, Z-index intervals, into k sequences $Z_1, \ldots, Z_k$ of z/k consecutive intervals each, where k is the largest value such that $k \leqslant z^{1/3}$ and $\log\log k$ is an integer. Note that this implies $k > z^{1/6}$.

We create an output stream for each sequence. We now push the edges of $\mathcal{F}$ into a k-distributor to distribute them to these output streams. When all edges have been pushed into the distributor, we process it top-down to flush all buffers. Next we distribute the segments in each stream further recursively. When $z \leqslant 8$, we choose $k = z$ and the output is written to one big output stream rather than a separate output stream for every single Z-index interval.

Lemma 3.8. *When $M \geqslant cB^2$, for a sufficiently large constant c, we can compute all intersections between the $O(n)$ edges of $\mathcal{F}$ and the quadtree subdivision of Lemma* 3.6 *cache-obliviously in $O(sort(n' \cdot (\lambda + \lambda^*)))$ I/Os, where $n' = n/\lambda^*$.*

Proof. We first analyse the cost of a single invocation of a k-distributor.

The space $M(k)$ needed by a k-distributor including all of its buffers is $O(k^2)$ for $\sqrt{k}$ input buffers of size $k^{3/2}$, plus $(\sqrt{k}+1)M(\sqrt{k})$ for the $\sqrt{k}$-distributors $D_0, \ldots, D_{\sqrt{k}}$; this recurrence solves to $M(k) = O(k^2)$. Hence, for a suitably small constant c' there is a $\bar{k}$ such that $\log\log\bar{k}$ is an integer, $c'M^{1/4} < \bar{k} \leqslant c'\sqrt{M}$, and a $\bar{k}$-distributor can be completely loaded and kept in memory, together with one block of its input stream as a read buffer and one block of each output stream as a write buffer.

Now, to analyze the number of I/Os needed to push the edges through a k-distributor, we consider two cases depending on whether $k = \bar{k}$ or $k > \bar{k}$. Let t_{in} be the number of edges in the input stream of the distributor, and t_{out} be the number of edges in the output stream.

- When $k = \bar{k}$ then the k-distributor fits completely into main memory. Hence, the number of I/Os for distributing the t_{in} edges to k streams of total size t_{out} is

$$O\left(\frac{t_{\text{in}}}{B} + \frac{\bar{k}^2}{B} + \frac{\bar{k}B}{B} + \frac{t_{\text{out}}}{B}\right) = O\left(\frac{\bar{k}^2}{B} + \bar{k} + \frac{t_{\text{out}}}{B}\right).$$

- When $k > \bar{k}$, we have to take into account that additional I/Os are needed to temporarily store segments in the buffers $B_1, \ldots, B_{\sqrt{k}}$ and to swap smaller distributors in and out of memory. We may assume that whenever we need a $\bar{k}$-distributor, we load it into memory completely (with all buffers inside it); when we need a larger distributor, we only load the constituent $\bar{k}$-distributors as needed, and at most one block at a time of the buffers between them. Note that each output segment is routed through at most $O(\log k/\log\bar{k}) = O(\log_{\bar{k}} k)$ $\bar{k}$-distributors, each of which (except the one at the top) has an input buffer larger than $\bar{k}^3$. The number of I/Os for swapping $\bar{k}$-distributors in and out of memory because their input buffers run full is therefore

$$O\left(\log_{\bar{k}} k \cdot \frac{t_{\text{out}}}{\bar{k}^3} \cdot \left(\frac{\bar{k}^2}{B} + \frac{\bar{k}B}{B}\right) + \frac{t_{\text{out}}}{B} \cdot \log_{\bar{k}} k\right) = O\left(\frac{t_{\text{out}}}{B} \cdot \log_{\bar{k}} k\right).$$

Finally, in the flushing phase every $\bar{k}$-distributor may be loaded once with an input buffer that contains less than $\bar{k}^3$ segments: there are $O(k/\bar{k})$ such distributors and each of them can be flushed in $O(\bar{k}^3/B)$ I/Os, for a total of $O(k\bar{k}^2/B)$ I/Os. The total cost of running a k-distributor with $k > \bar{k}$ is therefore

$$O\left(\frac{k\bar{k}^2}{B} + \frac{t_{\text{out}}}{B} \cdot \log_{\bar{k}} k\right).$$

Using $\bar{k} < k \leqslant z^{1/3}$ and $\log(M/B) = \Theta(\log M)$, the number of I/Os is thus

$$O\left(\frac{z}{B} + \frac{t_{\text{out}}}{B} \cdot \log_{\frac{M}{B}} \frac{z}{B}\right).$$

To analyse the full algorithm, note that z gets polynomially smaller with every level of recursion, and a stream will be distributed with all distributors in memory as soon as the output size, which is at most $O((\lambda + \lambda^*) \cdot z)$, drops below a certain constant times M. Similarly to the analysis by Brodal and Fagerberg [7] one can now show that over all levels of recursion, the second terms of the $O(z/B + (t_{\text{out}}/B)\log_{M/B}(z/B))$ bounds sum up to at most $O(sort(t_{\text{out}}))$ I/Os, where $t_{\text{out}} = O(n \cdot (\lambda + \lambda^*)/\lambda^*)$. Furthermore, this dominates the first terms of the bound. Thus the total cost of distributing n edges to the $n' = O(n/\lambda^*)$ Z-index intervals of the compressed quadtree subdivision is $O(sort(n' \cdot (\lambda + \lambda^*)))$ I/Os. □

Putting it all together. The complete algorithm to build a linear compressed quadtree on $\mathcal{F}$ is as follows:

1. Compute an initial compressed quadtree subdivision (Lemma 3.4).
2. Set $\lambda^* \leftarrow 1$.
3. Merge cells (Lemma 3.6).
4. Compute the intersection of $\mathcal{F}$ with the quadtree subdivision; however, as soon as a node v of $\mathcal{Z}$ is found to which at least $30\lambda^*|v|$ edges are written (where $|v|$ is the number of cells that descend from v), abort the computation, double λ^*, and try again from Step 3.
5. After completing the previous step successfully, sort all edge-cell intersections by the Z-indices of the cells, and put a B-tree on top of them. Each cell σ without any intersecting edges is merged with the cells that precede or follow it in the Z-order, up to a cell that stores an edge of the face of $\mathcal{F}$ that contains σ (see Section 2.3 for an explanation).

Lemma 3.9. *In $O(sort(n)\log\lambda)$ I/Os we can construct a linear compressed quadtree for $\mathcal{F}$ of $O(n)$ cells that each intersect $O(\lambda)$ edges of $\mathcal{F}$ and with $O(n)$ cell-edge intersections in total.*

Proof. Computing the initial compressed quadtree subdivision takes $O(sort(n))$ I/Os.

By Lemma 3.6, every cell intersects less than $24\lambda + 6\lambda^*$ edges; so, as soon as we have $\lambda^* \geqslant \lambda$, Step 4 cannot fail. Hence, Steps 3 and 4 are carried out at most $1 + \lceil \log\lambda \rceil$ times. Step 3 takes $O(sort(n))$ I/Os; Step 4 takes $O(sort(n))$ I/Os as well, since only $O(n)$ (copies of) edges are distributed: the algorithm is always aborted before the number of (copies of) edges being distributed would grow beyond $30\lambda^*$ times the number of quadtree cells, which is $O(n/\lambda^*)$. Thus, Steps 3 and 4 take $O(sort(n)\log\lambda)$ I/Os in total.

In the final step of the algorithm, $O(n)$ edge-cell intersections are sorted. □

Note that the $\log\lambda$-factor in the bound arises only from the need to repeat part of the algorithm several times until a suitable setting for λ^* is found. When the density λ is assumed to be a fixed constant, or when λ is known, one may immediately set λ^* equal to a constant or to λ and run the complete algorithm in $O(sort(n))$ I/Os.

3.4. Query operations

Map overlay can be done in exactly the same way as with the star-quadtree—see Section 2.4. The analysis is also similar; the only difference is the size of the data structure (now $O(n)$ instead of $O(n/\delta^2)$) and the number of I/Os needed to scan a single cell (now $O(scan(\lambda))$ instead of $O(scan(1/\delta))$). Thus, we obtain a bound of $O(scan(n+k))$ I/Os for the overlay operation.

For point location with a query point p, we also proceed as before, namely by searching with the Z-index $Z(p)$ in the B-tree. Let $[z, z')$ be the Z-index interval containing $Z(p)$. Then the search gives us a list L of $O(\lambda)$ segments whose key is z, which are exactly the segments intersecting the region of the plane corresponding to $[z, z')$. We pick a point p' inside this region on any of the segments in L, and construct a path from p to p' inside the region. The first segment in L hit by the path then tells us in which region p lies. (Constructing the path and testing for intersections can all be done in internal memory, so it does not influence the I/O-bound.) Hence, point location takes $O(scan(\lambda) + \log_B n)$ I/Os. The analysis of batched point location is the same as for star-quadtrees (Lemma 2.9); again we use the fact that the contents of a single cell always fit in memory (by our assumption that $M > c\lambda$ for a large enough constant c). Thus batched point location with k query points takes $O(scan(n) + sort(k))$ I/Os.

The analysis for range searching needs more care. In the last paragraph of the proof of Lemma 2.11 we used the fact that in a star-quadtree, all cells whose parents do not lie inside Q_ε must have diameter $\Omega(\varepsilon \cdot \mathrm{diam}(Q))$. However, in a guard-quadtree this is not true; we are working with the parent-child relations in the tree $\mathcal{T}$ from Lemma 3.6, where children can be much smaller than their parents. Nevertheless we can obtain a similar result:

Lemma 3.10. *The linear quadtree for subdivisions with density λ as described above can be used to report, for any constant-complexity query range Q and for any constant $0 < \varepsilon < 1$, the edges that intersect Q in $O(\frac{1}{\varepsilon} scan(\lambda) + \frac{1}{\varepsilon}(\log_B n) + scan(k_\varepsilon))$ I/Os, where k_ε is the number of edges that lie at distance at most $\varepsilon \cdot \mathrm{diam}(Q)$ from Q.*

Proof. As with linear quadtrees for triangulations, we use Lemma 2.10 to compute a collection Ψ_Q of $O(1/\varepsilon)$ disjoint canonical squares covering Q and staying inside Q_ε. For each square $\sigma \in \Psi_Q$, we search the B-tree with the starting point of its Z-order interval, and then scan from there until the interval's endpoint. Locating the starting points takes $O(\frac{1}{\varepsilon}\log_B n)$ I/Os. To analyse the amount of data scanned inside these intervals, we consider the tree $\mathcal{T}$ from Lemma 3.6.

First we count the number of nodes in $\mathcal{T}$ such that the regions of their parents in $\mathcal{T}$ lie completely inside Q_ε. Let $\mathcal{A}$ be the set of the highest ancestors of these cells whose regions lie completely inside Q_ε. From Lemma 3.6 it follows that the total number of cells in the subtrees rooted at the nodes in $\mathcal{A}$ is $O(n_\mathcal{A}/\lambda^*)$, where $n_\mathcal{A}$ is the number of bounding-box vertices of edges intersecting the regions of $\mathcal{A}$, which is $O(k_\varepsilon)$.

Second, we count the number of nodes in $\mathcal{T}$ such that the regions of their parents in $\mathcal{T}$ lie partially outside Q_ε. Divide these nodes into size classes S_i, with i taking integer values from $\lfloor \log(\sqrt{2} \cdot \varepsilon \cdot \mathrm{diam}(Q)) \rfloor$ to $\lfloor \log(\sqrt{2} \cdot \mathrm{diam}(Q_\varepsilon)) \rfloor$, where S_i contains the nodes *whose parents in* $\mathcal{T}$ have width 2^i and area 2^{2i}. In addition there may be a set of nodes L, whose parents have width larger than $\sqrt{2} \cdot \mathrm{diam}(Q_\varepsilon)$.

Denote by $M(i)$ the Minkowski sum of the boundary of Q_ε and a square of width 2^{i+1}. Because the boundary of Q has only a constant number of local extrema in x- and y-direction, $\mathrm{area}(M(i)) = O(\mathrm{diam}(Q)2^{i+1})$. Since each parent with children in S_i has area 2^{2i} and must lie completely within a horizontal and vertical distance of 2^i from the boundary of Q_ε, the number of parents with children in S_i is bounded by $\mathrm{area}(M(i))/2^{2i} = O(\mathrm{diam}(Q)/2^i)$. Since each parent has at most 12 children, we have $|S_i| = O(\mathrm{diam}(Q)/2^i)$. Summing over all size categories S_i gives $\sum |S_i| = O(1/\varepsilon)$ nodes.

The number of parents with children in L is at most four. This can be seen as follows. Let $\mathcal{C}$ be the set of at most four canonical squares that cover Q_ε and with width as small as possible but larger than $\sqrt{2} \cdot \mathrm{diam}(Q_\varepsilon)$. Now suppose there were a square ϕ among $\mathcal{C}$ such that there are two large parents σ and τ that both have a child leaf cell that covers part of ϕ. Then σ and τ must be nested, say $\tau \supset \sigma \supseteq \phi$, but this immediately contradicts that there is a leaf cell that is a child (not merely a descendant) of τ that covers part of ϕ.

In total we get $O(1/\varepsilon)$ parents, each with at most 12 leaf cells to be scanned among their children. Since by construction each cell intersects less than $30\lambda^*$ edges, the total number of I/Os needed to scan the cells in Q' is $O(\frac{1}{\varepsilon}(\log_B n) + k_\varepsilon + \frac{1}{\varepsilon} scan(\lambda^*))$; recall that $\lambda^* = O(\lambda)$. □

4. Conclusions

We have described new indexing structures for planar maps in the I/O-model of computation, which allow for efficient point location, map overlay, and range searching. Our algorithms work for planar maps that are fat triangulations or have low density.

The solution for triangulations is based on quadtrees, is considerably simpler than previous solutions, and even supports dynamic maintenance of the planar maps under updates.

The second construction, for low-density planar maps, is based on compressed quadtrees and is more complicated; however, our analysis gives a better dependency on the parameter that describes the input—in fact the density affects only the number of operations on the CPU and the (mild) lower bound on the memory size, but hardly the bounds on the number of I/Os for queries, and not at all the bounds on the size of the data structure. Unfortunately, it is not clear if there is any way to support updates; the difficulty is that in the low-density setting, a single edge can intersect many cells, even though there is a good bound on the total number of edge-cell intersections.

Both constructions use space linear in n. Which of our two structures would give the most compact data structure for triangulations in practice remains to be seen. Although the theoretical dependency on the minimum angle in the triangulation is better when the approach for low-density subdivisions is used, it will base the quadtree subdivision on many guards: a triangulation of n vertices has roughly $3n$ edges and, thus, roughly $6n$ extra bounding box vertices as guards. Moreover, we do not believe the bounds for our approach for triangulations in Section 2 are tight. Although it is easy to show that our approach will create $\Theta(n/\delta)$ cells in the worst case, and that a cell intersects $\Theta(1/\delta)$ triangles in the worst case, we have been unable to come up with a construction in which the total number of intersections between triangles and cells is $\omega(n/\delta)$. A related question is whether the star of a single vertex can actually generate $\Theta(1/\delta^3)$ cell-triangle intersections in the worst case; a better bound would immediately improve the memory requirements for the construction algorithm of the linear quadtree for triangulations.

A natural idea for building compressed quadtrees may be to simply keep splitting cells until each cell with its intersecting edges fits in a block. Our second construction gives some insight in why such an approach may actually yield an efficient

data structure: consider what happens if λ is a small constant, and we set λ^* in our construction equal to cB, for some constant c. We now obtain a linear-size data structure of $O(n/B)$ cells, each of which can be scanned in $O(1)$ I/Os—more or less the same effect as one would get with the 'natural approach' suggested above. A problem with this natural approach and with the argument just given is that we would not know how exactly donut cuts should be defined in this context, and we would not know how to construct the quadtree I/O-efficiently in a top-down way. Our bottom-up approach, based on a guarding set and merging underfull cells, solves these problems and thus provides a provably efficient implementation of the 'natural quad-B-tree' for planar maps.

Acknowledgement

The authors thank Sariel Har-Peled for his extensive contributions, and the referees for their comments.

Appendix A. Suppressing duplicate cell boundaries

In Section 3.3 we gave an algorithm to generate, in $O(sort(n))$ I/Os, a subdivision of the Z-order curve that corresponds to a compressed quadtree on the set $\mathcal{G}$ of guards. This algorithm generates multiple copies of cell boundaries, so that duplicates need to be identified and removed. Unfortunately we can only identify duplicates after writing them to disk and sorting them. Here we show how to avoid generating duplicates.

To this end, we run the algorithm while maintaining an I/O-efficient stack of nested squares whose subdivisions were already output. Initially the stack is empty. We process $\mathcal{G}$ in Z-order, doing the following for each pair of consecutive, non-coinciding guards p and q. We pop all squares that do not contain q from the stack (let σ be the last square popped), and we compute $\mathrm{lca}(p,q)$. Let τ be the square that is now on top of the stack. If $\mathrm{lca}(p,q)=\tau$, we do not do anything. Otherwise we output the Z-indices that bound and separate the children of $\mathrm{lca}(p,q)$, excluding Z-indices that bound and separate children of σ or τ, and we push $\mathrm{lca}(p,q)$ on the stack. After handling all points of $\mathcal{G}$, we sort the Z-indices output.

Lemma A.1. *The above algorithm generates a subdivision of the Z-order curve that corresponds to a compressed quadtree on $\mathcal{G}$ in $O(sort(n))$ I/Os.*

Proof. The above algorithm only suppresses Z-indices that were already output, so it still outputs all the different Z-indices that are output by the algorithm of Lemma 3.4.

To see that the new algorithm does not output any Z-indices twice, we need to argue that it always suffices to check the stack for squares that were handled before. More precisely, we claim that whenever we output the Z-indices that bound and separate the children of a square σ, we put that square on the stack, and we will only remove a square from the stack when it does not need to be checked against Z-indices that may be considered for output in the future anymore. The first part of this claim follows directly from the algorithm. For the second part, when handling a pair of guards (p,q), we consider outputting the Z-indices that bound and separate the children of $\mathrm{lca}(p,q)$. Let ϕ be the child of $\mathrm{lca}(p,q)$ that contains p, and let σ be the largest square on the stack such that $\sigma\subseteq\phi$. Note that $p\in\sigma$. Since q comes after ϕ in the Z-order, we will never encounter a lowest common ancestor equal to or inside ϕ anymore. Hence we will never consider outputting Z-indices properly inside ϕ anymore. Outputting the Z-indices that bound ϕ in the future will be prohibited by pushing $\mathrm{lca}(p,q)=\mathrm{parent}(\phi)$ on the stack. Therefore we can safely remove ϕ and all cells included in ϕ from the stack. One can now verify that the cells that are kept on the stack always form a nested sequence, and that one only needs to check the largest cell properly inside $\mathrm{lca}(p,q)$ and the smallest cell that contains $\mathrm{lca}(p,q)$ to see if any Z-indices that bound or separate the children of $\mathrm{lca}(p,q)$ were output already. □

References

[1] A. Aggarwal, J.S. Vitter, The input/output complexity of sorting and related problems, Commun. ACM 31 (1988) 1116–1127.
[2] L. Arge, T. Mølhave, N. Zeh, Cache-oblivious red–blue line segment intersection, in: Proc. 16th Annu. European Sympos. Algorithms, 2008, pp. 88–99.
[3] L. Arge, J. Vahrenhold, I/O-efficient dynamic planar point location, Comput. Geom. Theory Appl. 29 (2) (2004) 147–162.
[4] L. Arge, D.E. Vengroff, J.S. Vitter, External-memory algorithms for processing line segments in geographic information systems, Algorithmica 47 (1) (2007) 1–25.
[5] M. Bender, R. Cole, R. Raman, Exponential structures for efficient cache-oblivious algorithms, in: Proc. 29th Internat. Colloq. Automata Lang. Prog., in: Lecture Notes in Computer Science, vol. 2380, Springer-Verlag, Berlin, 2002, pp. 195–207.
[6] M.A. Bender, E.D. Demaine, M. Farach-Colton, Cache-oblivious B-trees, SIAM J. Comput. 35 (2) (2005) 341–358.
[7] G.S. Brodal, R. Fagerberg, Cache oblivious distribution sweeping, in: Proc. 29th International Colloquium on Automata, Languages, and Programming, in: Lecture Notes in Computer Science, vol. 2380, Springer-Verlag, Berlin, 2002, pp. 426–438.
[8] G.S. Brodal, R. Fagerberg, R. Jacob, Cache oblivious search trees via binary trees of small height, in: Proc. 13th ACM–SIAM Symp. Discrete Algorithms, 2002, pp. 39–48.
[9] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, MIT Press/McGraw-Hill, Cambridge, MA, 2001.
[10] A. Crauser, P. Ferragina, K. Mehlhorn, U. Meyer, E. Ramos, Randomized external-memory algorithms for some geometric problems, Comput. Geom. Theory Appl. 11 (3) (June 2001) 305–337.
[11] M. de Berg, Improved bounds on the union complexity of fat objects, in: Proc. 25th Conf. Found. Soft. Tech. Theoret. Comput. Sci., 2005, pp. 116–127.
[12] M. de Berg, O. Cheong, M. van Kreveld, M. Overmars, Computational Geometry: Algorithms and Applications, Springer-Verlag, 2008.
[13] M. de Berg, M.J. Katz, A.v. Stappen, J. Vleugels, Realistic input models for geometric algorithms, Algorithmica 34 (2002) 81–97.

[14] M. de Berg, M. Streppel, Approximate range searching using binary space partitions, Comput. Geom. Theory Appl. 33 (3) (February 2006) 139–151.
[15] U. Finke, K. Hinrichs, Overlaying simply connected planar subdivisions in linear time, in: Proc. 11th Annu. ACM Symp. Comput. Geom., 1995, pp. 119–126.
[16] M. Frigo, C.E. Leiserson, H. Prokop, S. Ramachandran, Cache-oblivious algorithms, in: Proc. 40th Annu. IEEE Symp. Found. Comput. Sci., 1999, pp. 285–298.
[17] I. Gargantini, An effective way to represent quadtrees, Commun. ACM 25 (12) (1982) 905–910.
[18] M.T. Goodrich, J.-J. Tsay, D.E. Vengroff, J.S. Vitter, External-memory computational geometry, in: Proc. 34th Annu. IEEE Symp. Found. Comput. Sci., 1993, pp. 714–723.
[19] H. Haverkort, M. de Berg, J. Gudmundsson, Box-trees for collision checking in industrial installations, Comput. Geom. Theory Appl. 28 (2–3) (2004) 113–135.
[20] G.R. Hjaltason, H. Samet, Speeding up construction of pmr quadtree-based spatial indexes, VLDB Journal 11 (2002) 190–237.
[21] H. Samet, Foundations of Multidimensional and Metric Data Structures, Morgan-Kaufman, San Fransisco, 2006.
[22] M. Streppel, K. Yi, Approximate range searching in external memory, in: Proc. 18th Int. Symp. on Algorithms and Computation, 2007, pp. 536–548.